Ausnutzung verdeckter Kanäle am Beispiel eines Web-Servers

von

Dipl.-Math. Hans-Georg Eßer

Diplomarbeit

in Informatik



angefertigt am
Lehr- und Forschungsgebiet Informatik IV
RWTH Aachen

Erstprüfer: Prof. Dr.-Ing. Felix C. Freiling geb. Gärtner
Zweitprüfer: Prof. Dr. rer. nat. Peter Rossmanith

21. Februar 2005

Ausnutzung verdeckter Kanäle am Beispiel eines Web-Servers

von Hans-Georg Eßer,
h.g.esser@gmx.de

Diplomarbeit im Fach Informatik,
Lehr- und Forschungsgebiet Informatik IV,
Rheinisch-Westfälische Technische Hochschule Aachen,
21. Februar 2005

Zusammenfassung

Diese Diplomarbeit beschreibt die Implementierung eines unidirektionalen verdeckten Kanals (*covert channel*) zwischen einem modifizierten Web-Server (Apache) und einem HTTP-Proxy-Server. Es wird ein verdeckter Zeitkanal (*covert timing channel*) aufgebaut, der die reguläre Kommunikation zwischen Apache-Server und Proxy nutzt und einzelne übertragene Pakete zeitlich verzögert. Auf Empfängerseite wird die Verzögerung gemessen und ausgewertet. Dieser Kanal kann zur unerkennbaren Übertragung von Informationen verwendet werden.

Nach einer Einführung in die Grundlagen der Informationstheorie und verdeckter Kanäle werden der Proxy-Server und die Modifikationen am Apache-Quellcode beschrieben. Kapazitätsmessungen zeigen, wie nützlich der verdeckte Kanal ist.

Bei den Messungen zeigt sich, dass die Fehlerrate mit sinkender Verzögerungszeit exponentiell steigt; der Einsatz eines Fehlerkorrekturverfahrens verbessert die Ergebnisse nicht wesentlich.

Schlüsselbegriffe: Verdeckter Kanal, Verdeckter Zeitkanal, Apache, Web-Server, Sicherheit, Kanalkapazität.

Exploiting covert channels using a Web server

by Hans-Georg Eßer,
h.g.esser@gmx.de

Thesis for the degree of Diplom-Informatiker (Dipl.-Inform.),
Laboratory of Dependable Distributed Systems,
University of Technology (RWTH) Aachen, 21st February 2005

Abstract

This thesis presents an implementation of a unidirectional covert channel that exists between a modified Web server (Apache) and an HTTP proxy server. We establish a covert timing channel that is embedded in the regular channel from client through proxy to server, by adding artificial delays in the data flow from server to proxy. This channel can be used to transport hidden information.

An introduction to the basic concepts of information theory and covert communication is followed by a description of the proxy server and the changes made to the Apache source code. Capacity measurements show how useful the covert channel is.

Tests show that the error rate of the covert channel grows exponentially as the delays are reduced. Using an error correction code leads to no substantial reduction of the error rate.

Keywords: Covert channel, Timing channel, Apache, Web server, Security, Channel capacity.

Dankwort

Herrn Professor Dr.-Ing. Felix Freiling geb. Gärtner danke ich für die Möglichkeit, als in München berufstätiger Diplomand eine Diplomarbeit am Lehr- und Forschungsgebiet für Informatik IV zu schreiben, sowie für seine Anmerkungen zu mehreren Vorabversionen dieser Arbeit.

Herrn Professor Dr. rer. nat. Peter Rossmanith danke ich für die Übernahme des Korreferats.

Ich danke auch meinem Kollegen Dr.-Ing. Achim Leitner, mit dem ich das Verhalten des TCP-Stacks diskutiert habe.

Für das äußerst sorgfältige Korrekturlesen dieser Arbeit danke ich Heike Jurzik, M.A.

LaTeX

Diese Arbeit wurde mit dem Textsatzsystem TeX von D. E. Knuth und dem Makropaket LaTeX von L. Lamport geschrieben. Für einige Spezial-Features wurde das KOMA-Skript-Paket verwendet.

Eine unverzichtbare Hilfe bot der *LaTeX-Begleiter* [GMS95], der Ordnung in die verwirrende Vielfalt des LaTeX-Paketes brachte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

> Denn die einen stehn im Dunkeln, und die andern stehn im Licht.
> Und man sieht nur die im Lichte, die im Dunkeln sieht man nicht.
>
> *(Bertold Brecht, Dreigroschenoper 1928)*

Verdeckte Kommunikation hat in den vergangenen Jahren einen höheren Stellenwert bei der Untersuchung von sicherheitskritischen Rechnernetzen gewonnen: Das gestiegene Bewusstsein für die Gefahr von Angriffen hat dazu geführt, dass Trivialangriffe nur noch eingeschränkt möglich sind – in der Folge greifen Angreifer auf fortgeschrittenere Techniken zurück.

Diese Arbeit behandelt Grundlagen der Kommunikation über verdeckte Kanäle: Ein Kommunikationskanal wird als verdeckt betrachtet, wenn er unvorhergesehen ist, in dem Sinne, dass nicht mit Kommunikation über diesen Kanal gerechnet wurde, als der betroffene Rechner bzw. das betroffene Netzwerk eingerichtet wurde.

Verdeckte Kommunikation ist unerwünscht, da sie Sicherheitsmechanismen untergräbt: Regulierung und Überwachung von Kommunikation betrachten nur die „regulären“, also nicht verdeckten Kanäle.

1.1 Motivation

Ein *Trojaner* oder *trojanisches Pferd* ist nach den meisten Definitionen ein Programm, das Anwender installieren, weil es interessante Funktionen verspricht – diese hat es dann auch in der Regel, besitzt aber außerdem nicht dokumentierte und unerwünschte Zusatzfunktionen, die es beispielsweise dem Autor des Trojaners erlauben, einen Rechner, auf dem es installiert wurde, für eigene Zwecke zu nutzen.

Beliebt ist etwa die Verwendung großer Mengen privater PCs für „DDOS-Attacken“ (*distributed denial of service attacks*): Bei einer DDOS-Attacke wird ein Zielrechner angegriffen, indem zu einem festgelegten Zeitpunkt Anfragen von sehr vielen Rechnern an den Zielrechner geschickt werden. Dazu schickt der Angreifer an alle von ihm kontrollierten Rechner beispielsweise den Befehl, an einen Zielrechner eine

längere Reihe von ICMP-Echo-Paketen (definiert in RFC 792 [Pos81a]) zu schicken. Von einem einzigen Rechner ausgehend wäre dies für den Zielrechner nicht belastend; hat der Angreifer aber Hunderte oder Tausende Maschinen über den Trojaner unter Kontrolle gebracht, kann der Zielrechner unter dieser Last zusammenbrechen: Er kann dann reguläre Anfragen nicht mehr beantworten.

Trojaner erlauben dem Programmierer (im Folgenden Angreifer genannt) nach ihrer Installation die Anmeldung auf dem betroffenen Rechner oder dessen Fernwartung. Dazu öffnen sie meist einen TCP-Port, über den die Kommunikation abgewickelt wird. Beim Start einer DDOS-Attacke wird dann allen erreichbaren kontrollierten Rechnern der auszuführende Befehl geschickt.

Auf der Seite des betroffenen Rechners kann diese unerwartete Kommunikation festgestellt oder durch eine Firewall verhindert werden. Um dennoch Kontakt mit dem betroffenen Rechner halten zu können, sind also andere Übertragungswege nötig.

Tunneln

Prinzipiell kann der Trojaner ein beliebiges Programm, das auf dem betroffenen PC häufig verwendet wird und das Internet-Verbindungen aufbaut, durch ein angepasstes Programm mit gleicher Funktion ersetzen, das dann bei Nutzung zusätzlich mit einem Server des Angreifers kommuniziert.

Meldet sich der Trojaner beispielsweise über den – auch bei Einsatz einer Firewall meist offenen – HTTP-Port 80 beim Server des Angreifers, kann er auf dem Weg vom Server Anweisungen erhalten; zudem kann der Trojaner schon bei der ersten Anfrage Informationen über den Status des Rechners übermitteln. Die beschriebene Vorgehensweise kann als Tunneln betrachtet werden: Es gibt Programme, die beispielsweise eine reguläre Shell-Sitzung über einen HTTP-Tunnel ermöglichen, z. B. *revsh* [rev]. (Typischer ist das vollständige Tunneln einer anderen Verbindung, wie es beispielsweise das Programm *httptunnel* [Htt] ermöglicht; siehe Abbildung 1.1.)

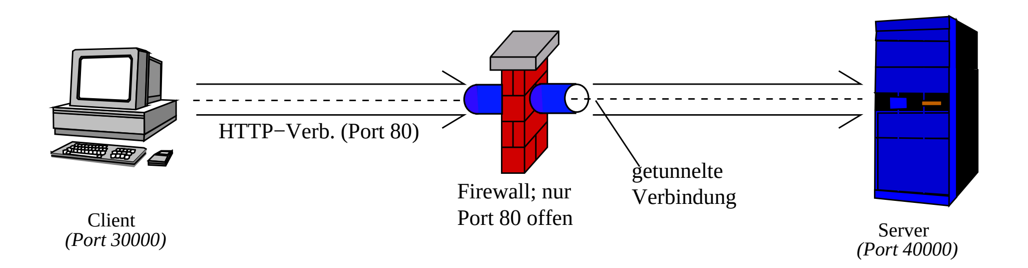


Abbildung 1.1: Tunneln einer Verbindung über Port 80.

Das Programm des Angreifers nutzt hier die Tatsache aus, dass der genutzte Port 80 von Firewalls als sicher betrachtet und nicht blockiert wird. Bei einer Untersuchung des Netzwerkverkehrs würde allerdings auffallen, dass Verbindungen aufgebaut wurden, die nicht bewusst vom Benutzer des Rechner initiiert wurden; außerdem wären die übertragenen Inhalte auffällig: Statt Web-Seiten überträgt der Trojaner andere Daten.

Verdeckte Kommunikation

Soll die Kommunikation dauerhaft unerkannt bleiben, muss sie über weitere Techniken verborgen werden: Damit kommen verdeckte Kanäle ins Spiel. Die Vorgabe ist nun, keine (sichtbare) zusätzliche Kommunikation stattfinden zu lassen – das einfache Versenden und Empfangen von Daten ist damit ausgeschlossen. Stattdessen muss reguläre Kommunikation auf eine Weise verändert werden, die sich mit herkömmlichen Methoden nicht beobachten lässt. Häufig bedeutet das eine künstliche Variation der Reaktionszeiten, die vom Programm des Angreifers beobachtet und geeignet interpretiert werden.

Bei einer Überprüfung des Netzwerkverkehrs sind dann nur unverdächtige Übertragungen sichtbar, die vom Anwender initiiert wurden – das Timing-Verhalten wird nicht beachtet.

Wenn beispielsweise auf dem betroffenen Rechner der Web-Browser durch ein funktionsgleiches Programm ersetzt wird, das als Standard-Proxy für alle Verbindungen einen entsprechend modifizierten Proxy-Server auf dem Rechner des Angreifers verwendet, kann jeder Aufruf einer Web-Seite für die Zwecke des Trojaners eingesetzt werden. (Bei einer Untersuchung des Netzwerkverkehrs wären die übertragenen Daten unauffällig; allerdings würde man hier feststellen, dass alle Web-Seiten über einen fremden Proxy-Server geholt werden.)

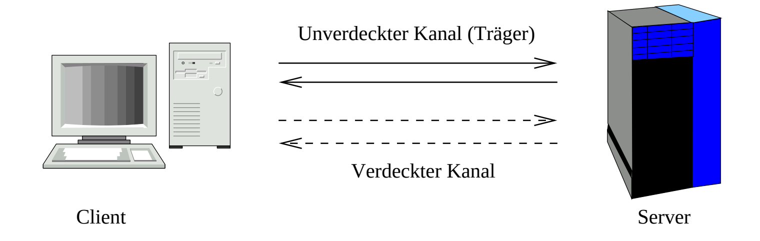


Abbildung 1.2: Verdeckter Kanal mit zugehörigem Trägerkanal.

1.2 Beschreibung der Aufgabenstellung

Ziel dieser Arbeit ist es, anhand einer Beispielimplementation das Risiko verdeckter Kommunikation aufzuzeigen und durch Messungen zu überprüfen, ob der implementierte verdeckte Kanal eine ausreichende Kapazität hat, um „nützlich“ zu sein.

Dazu werden ein WWW-Proxy-Server so programmiert und ein WWW-Server so verändert, dass sie neben der regulären Kommunikation einen verdeckten Kanal aufbauen.

Im Anschluss wird die Kapazität des so eingeführten verdeckten Kanals analysiert und gegebenenfalls optimiert.

1.3 Ergebnisse

Durch Anpassung des Apache-Servers und mit Hilfe verschiedener Zusatzprogramme, die in Python programmiert wurden, konnte der HTTP-Kommunikation ein verdeckter Kanal hinzugefügt werden. Die Datenrate auf diesem Kanal ist gering (1 Bit verdeckte Daten pro 4 KByte reguläre Daten – das sind ca. 0,003 %), und es ist eine hohe zeitliche Verzögerung nötig, um die Fehlerwahrscheinlichkeit auf dem verdeckten Kanal klein zu halten. Messungen mit verschiedenen Verzögerungen ergaben, dass die Fehlerquote exponentiell wächst, wenn die Verzögerung verringert wird.

Der Einsatz eines Fehlerkorrekturverfahrens konnte die Fehlerquote in einigen Fällen reduzieren, brachte aber keine wesentliche Verbesserung. Auch mit Fehlerkorrektur blieb der exponentielle Zusammenhang zwischen Verzögerung und Fehlerrate erhalten.

Ohne Fehlerkorrektur wurden selbst bei einer durchschnittlichen Verzögerung von 0,25 s pro 4-KByte-Block Nutzdaten (das entspricht einer Versiebenfachung der Transferzeit für eine 5-MByte-Datei von 42 s auf 322 s) geringfügig fehlerhafte Daten empfangen. (Als durchschnittliche Verzögerung wird hier die halbe Verzögerungszeit verzögerter Pakete bezeichnet, da im Mittel gleich viele verzögerte und unverzögerte Pakete verschickt werden.)

Mit Hilfe der Fehlerkorrektur wurden fehlerfreie Übertragungen bei durchschnittlichen Verzögerungen ab 0,2 s erreicht. Bei sehr kurzen Verzögerungen führte die Fehlerkorrektur kaum zu Verbesserungen, sondern teilweise sogar zu Verschlechterungen. Für durchschnittliche Verzögerungen ab 0,075 s (und höher) waren die Fehlerraten mit Fehlerkorrektur durchgehend niedriger als ohne.

Obige Ergebnisse gelten für die Übertragung über eine Internet-Verbindung. Im lokalen Netzwerk waren die Ergebnisse erheblich besser: Hier reichte eine durchschnittliche Verzögerung von 0,05 s aus, um fehlerfreie Übertragungen zu erreichen; auch mit 0,015 s und 0,005 s durchschnittlicher Verzögerung blieb die Fehlerrate mit 0,13 % sehr klein.

Bei rein lokalen Tests (bei denen Server und Client auf dem gleichen Rechner liefen) traten gar keine Fehler auf.

Das Internet-Szenario ist aber realitätsnäher und hat darum besondere Beachtung erfahren.

1.4 Gliederung der Arbeit

Ein Grundlagenteil (Kapitel 2) stellt die Definition verdeckter Kanäle vor und erklärt anhand von Beispielen den Unterschied zwischen Ressourcenkanal (*resource channel*) und Zeitkanal (*timing channel*). Elementare Definitionen aus der Informationstheorie schließen sich daran an; hier geht es vor allem um die Kanalkapazität. Die Begriffe werden auf verdeckte Kanäle übertragen. Es folgen eine Vorstellung von

Techniken zur Fehlererkennung und -korrektur sowie eine Beschreibung des HTTP-Protokolls inklusive Grundlagen von TCP/IP und die Definition einiger statistischer Begriffe.

Anschließend an diese einführenden Betrachtungen beschreibt die Arbeit eine konkrete Implementation eines verdeckten Kanals (Kapitel 3). Ein modifizierter Web-Server kommuniziert verdeckt mit einem HTTP-Proxy, während der Proxy „normale" Web-Anfragen an den Server richtet, die dieser beantwortet. Als Kanal wird die Reaktionszeit des Web-Servers verwendet: So entsteht ein Zeitkanal. Diese Reaktionszeiten sind aber nicht ausschließlich vom Server beeinflusst, sondern können auch unter veränderter Netzlast schwanken; daher wird die verdeckte Kommunikation um ein Fehlerkorrekturverfahren erweitert.

Interessant ist neben der reinen Realisierung die Betrachtung der durchschnittlich erreichbaren Datenrate auf dem verdeckten Kanal in Relation zur Datenrate der „normalen" Kommunikation zwischen Proxy und Server über das HTTP-Protokoll. Statistische Analysen und eine Bewertung der Nützlichkeit liefert Kapitel 4.

Kapitel 5 fasst die Ergebnisse zusammen und betrachtet mögliche Erweiterungen: Die beschriebene Kommunikation erfolgt nur in einer Richtung, nämlich vom Web-Server zum Proxy. Ein verdeckter Rückkanal (beispielsweise zur Empfangsbestätigung) ist eine sinnvolle Ergänzung; die Verwendung zusätzlicher Methoden zum Verbergen von Informationen im Datenstrom (etwa über Steganographie) eine weitere. Kapitel 6 stellt verwandte Arbeiten vor.

Anhang A enthält die Originalfassung der Aufgabenstellung; ein Benutzerhandbuch für die entwickelte Software bietet Anhang B. Im Anhang C sind die wichtigsten Code-Teile der Implementation mit Kommentaren abgedruckt, Anhang D beschreibt die zu dieser Arbeit gehörende Begleit-CD, und Anhang E betrachtet die Kommunikation zwischen Web-Server und Client im Detail.

1.5 Voraussetzungen an den Leser

Diese Arbeit besteht zu einem großen Teil aus Beschreibungen, wie die vorgestellten Verfahren implementiert wurden; dazu werden auch verschiedene Listings abgedruckt. Für das Verständnis der Code-Teile sind Grundkenntnisse in den Programmiersprachen C und Python [Lut96, Bea99] notwendig.

Grundlagen der (Unix-) Netzwerkkommunikation über Sockets [Ste00, CSo04] werden ebenfalls vorausgesetzt.

Ferner sollten Grundlagenkenntnisse in linearer Algebra [Lan70] und Wahrscheinlichkeitsrechnung [MP90] vorhanden sein; die nötigen Konzepte werden aber in Kurzform vorgestellt.

Kapitel 2

Grundlagen

Dieses Kapitel liefert die theoretischen Grundlagen zu verdeckten Kanälen, stellt Konzepte der Informationstheorie vor und vergleicht verdeckte Kommunikation mit den verwandten Techniken der Steganographie und Kryptographie. Außerdem werden ein einfaches Fehlerkorrekturverfahren und die Grundlagen des HTTP-Protokolls beschrieben.

2.1 Einleitung

Kryptographie und verdeckte Kommunikation haben ähnliche Ziele, dabei aber ein wesentliches Unterscheidungsmerkmal: Während Kryptographie Kommunikation verschlüsselt, so dass nur der rechtmäßige Empfänger eine Botschaft entschlüsseln kann, geht verdeckte Kommunikation einen Schritt weiter und verbirgt vor Außenstehenden die Tatsache, dass überhaupt Kommunikation stattfindet.

Beide Methoden können auch verbunden werden, so dass selbst, wenn verdeckte Kommunikation erkannt wurde, immer noch die Vertraulichkeit der Nachrichten gesichert ist.

In der Eigenschaft, Kommunikation zu verbergen, sind verdeckte Kanäle auch mit Steganographie verwandt: Dort werden geheime Botschaften in anderen Nutzdaten, zum Beispiel Bildern oder Audiodateien versteckt. Ein einfacher steganographischer Ansatz ist etwa, in einer Datei in jedem n-ten Byte das niedrigste Bit zu verändern, so dass sich aus den angepassten Bits die Nachricht zusammensetzen lässt. Einen komplexeren, graphentheoretischen Ansatz, bei dem in einem Bild möglichst wenige Pixel verändert werden und die Anteile der Farben möglichst stabil bleiben, verfolgt das Open-Source-Programm *steghide* [Het02]. Auch Steganographie kann mit Kryptographie verbunden werden.

Das folgende Kapitel 2.2 stellt verdeckte Kanäle vor, Kryptographie und Steganographie sind Gegenstand von Kapitel 2.3. Kapitel 2.4 beschreibt einige Anwendungsgebiete verdeckter Kommunikation, und Kapitel 2.5 führt in die Informationstheorie ein und behandelt auch die Kapazität verdeckter Kanäle.

Fehlererkennung und -korrektur werden eingesetzt, wenn der verwendete Kanal gestört ist – Korrekturverfahren können dabei eine fehlerhaft übertragene Nachricht in

den Ausgangszustand zurückversetzen, wenn eine vorgegebene Fehlerquote nicht überschritten wird.

Kapitel 2.6 ist ein Exkurs in die lineare Algebra, der für einen Teil des folgenden Kapitels 2.7 nötig ist, das sich mit Fehlererkennung und -korrektur beschäftigt.

Kapitel 2.8 behandelt das für Verbindungen zwischen Web-Servern und Clients eingesetzte Protokoll HTTP und die TCP/IP-Grundlagen (insbesondere Fragmentierungsfragen), und Kapitel 2.9 definiert einige grundlegende Begriffe aus der Statistik.

2.2 Verdeckte Kanäle

Dieser Abschnitt beschreibt, wie sich verdeckte Kommunikation von normaler Kommunikation unterscheidet. Es werden Ressourcenkanäle (*resource channels*) und Zeitkanäle (*timing channels*) definiert und mehrere Beispiele für diese Kanaltypen sowie Kombinationen gegeben. Außerdem werden verdeckte Kanäle mit anderen Techniken der Geheimhaltung, insbesondere mit Kryptographie und Steganographie, verglichen.

Definition

Ein **Kanal** (Kommunikationskanal) ist ein Übertragungsweg für **Nachrichten** (Informationen): Der Kanal verbindet **Sender** und **Empfänger** dieser Nachrichten.

Diese Arbeit behandelt ausschließlich Kanäle, die zwischen zwei Programmprozessen (auf verschiedenen Rechnern oder auf derselben Maschine) existieren – klassische Kanäle, etwa Funkübertragungen (Radio, Fernsehen, Funkgeräte) und Telefonverbindungen, werden hier nicht betrachtet, aber auch für diese allgemeinere Kanaldefinition kann man das im Folgenden vorgestellte Konzept des verdeckten Kanals betrachten.

Definition

Ein **verdeckter Kanal** (*covert channel*) ist ein Kommunikationskanal zwischen zwei Prozessen auf einem einzelnen Rechner (lokal) oder zwischen vernetzten Rechnern, der zustande kommt, indem „ein Computersystem-Mechanismus in einer unerwarteten Weise genutzt wird, um ein Mittel zur Verfügung zu stellen, durch das Information zu einem nicht autorisierten Individuum fließen kann." (Diese Definition wurde aus dem Buch von Amoroso [Amo94] übernommen.)

Verdeckte Kanäle wurden erstmals von Lampson in seinem 1973 erschienenen Artikel „A Note on the Confinement Problem" [Lam73] beschrieben.

Entscheidend für die Frage, welcher Kanal verdeckt ist, ist der vorgegebene Sicherheitskontext. Man geht davon aus, dass Systembenutzer in verschiedene Sicherheitsstufen eingeordnet sind und Informationen von Anwendern auf einer höheren Stufe solchen auf einer niedrigeren nicht zugänglich gemacht werden dürfen. Die Netzwerk- oder Systemadministratoren haben die Aufgabe, Informationsfluss zu verhindern, der diese Sicherheitsstandards verletzen würde. In dieses Konzept passt auch ein Angriff von außen, wenn man externe, anonyme Kommunikationspartner als Anwender mit der niedrigsten Sicherheitsstufe ansieht.

Das folgende Beispiel zeigt einen verdeckten Kanal, der durch geeignete Wahl von Dateinamen in einem temporären Verzeichnis entsteht.

Auf einem Unix-System herrschen folgende Einschränkungen: Benutzer der Gruppe *untrusted* können Dateien nur mit den Rechten *600* (`rw-------`) erzeugen, Aufrufe von `chmod` sind verboten, auch entsprechende Systemroutinen können nicht über andere Programme angesprochen werden. Dadurch soll verhindert werden, dass Benutzer dieser Gruppe Daten austauschen.

Beispiel

Über einen Befehl der Form

```
touch /tmp/Nachricht-für-alle-lesbar
```

kann dennoch jeder Benutzer Nachrichten übermitteln – dabei wird vorausgesetzt, dass alle Benutzer Lese- und Schreibzugriff auf das temporäre Verzeichnis `/tmp` besitzen.

Ein **verdeckter Systemkanal** liegt vor, wenn ein Prozess bzw. ein Anwender Informationen über das System erlangt, die ihm regulär nicht zugänglich wären. Der Kommunikationspartner ist hier also der Rechner oder das Netzwerk selbst.

Definition

Es folgt ein Beispiel für einen verdeckten Systemkanal:

Bei der Anmeldung an einem Unix-System werden vom Login-Prozess Benutzername und -passwort erfragt. Ist der Benutzername dem System bekannt, wird das eingegebene Passwort verschlüsselt und mit dem verschlüsselt gespeicherten Passwort verglichen. Abhängig vom Vergleich wird der Login erlaubt oder verweigert. Ist der Benutzername unbekannt, findet kein Passwortvergleich statt. Die Auswertung der Login-Daten dauert damit – abhängig vom Vorhandensein des Accounts – unterschiedlich lange. Auf diesem Weg kann durch Login-Versuche ermittelt werden, welche Benutzer-Accounts auf dem Rechner vorhanden sind, sofern das Login-Programm nicht durch eine künstliche Verzögerung dafür sorgt, dass die Antwortzeit immer gleich ist.

Beispiel

Ein **regulärer Kanal** (*overt channel*) ist ein Kanal, der nicht verdeckt ist.

Definition

Wenn ein verdeckter Kanal stets in Verbindung mit einem bestimmten regulären Kanal auftritt, also die Nutzung dieses Kanals voraussetzt, wird der reguläre Kanal als **Transportkanal** für den verdeckten Kanal bezeichnet.

Definition

Ein Kanal (egal, ob regulär oder verdeckt) heißt **unidirektional**, wenn die Informationen nur in eine Richtung fließen können, es also genau einen Sender und einen Empfänger gibt. Er heißt **bidirektional**, wenn Informationen in beide Richtungen fließen können. Sind zwei separate unidirektionale Kanäle von A nach B und von B nach A vorhanden, bilden beide zusammen einen bidirektionalen Kanal.

Definition

2.2.1 Bell-LaPadula-Modell

Das Bell-LaPadula-Modell [BL73], u. a. beschrieben im Buch von Amoroso [Amo94, Kapitel 9], führt verschiedene Sicherheitsebenen ein, auf denen sich Subjekte und Objekte befinden können. Subjekte sind z. B. Benutzerprozesse, Objekte z. B. Dateien. Genauer definiert Amoroso [Amo94]:

> A *subject* will be defined as an active computer system entity that can initiate requests for resources and utilize these resources to complete computing tasks. In the context of an operating system, subjects are typically the processes or tasks that operate on behalf of the users of the system. [. . .]
>
> An *object* will be defined as a passive computer system repository that is used to store information. In the context of an operating system, objects are typically the files and directories that maintain user information on the system. [. . .]

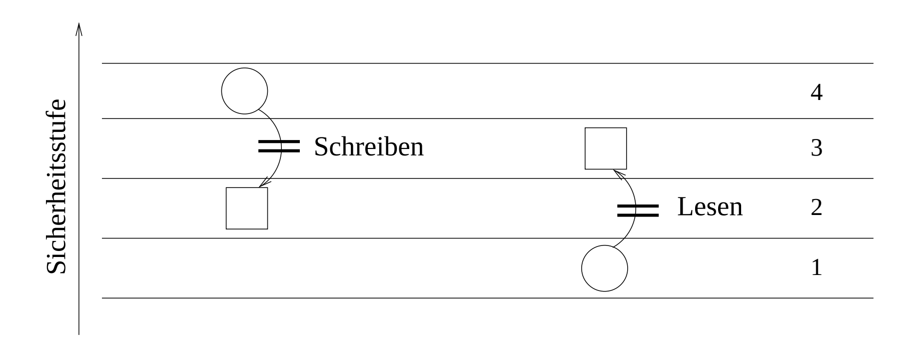


Abbildung 2.1: Die zwei Grundregeln des Bell-LaPadula-Modells.

Im Wesentlichen führt das Bell-LaPadula-Modell zwei Regeln ein: Subjekte dürfen keine Objekte niedrigerer Stufe schreiben und keine Objekte höherer Stufe lesen. (Abbildung 2.1; Subjekte sind als Kreise, Objekte als Quadrate dargestellt.) Diese Regeln heißen *No write down* (NWD-Regel) und *No read up* (NRU-Regel).

Ein Betriebssystem, das diese Regeln umsetzt, verbietet es

- einem normalen Benutzer, vertrauliche Daten zu lesen,
- und einem als Geheimnisträger klassifizierten Benutzer, Daten zu schreiben, die eine normale Klassifizierung haben.

Formalisiert haben diese Regeln die Form

$$\text{NRU}: \quad \forall s \in S, o \in O : allow(s, o, \text{read}) \iff label(s) \geq label(o)$$
$$\text{NWD}: \quad \forall s \in S, o \in O : allow(s, o, \text{write}) \iff label(o) \geq label(s)$$

wobei S die Menge der Subjekte und O die Menge der Objekte sind und $label$ eine Abbildung von $S \cup O$ auf eine Menge von Labels ist, die über $\geq$ vergleichbar sind.

In Abbildung 2.1 könnten beispielsweise die Labels 1, 2, 3 und 4 für die vier sichtbaren Bereiche vergeben werden, $\geq$ ist dann die normale Größer-Gleich-Relation auf den natürlichen Zahlen.

Die Informationsflussdarstellung im Bell-LaPadula-Modell wird in vielen Veröffentlichungen zu verdeckten Kanälen verwendet, da verdeckte Kanäle stets die Eigenschaft haben, eine der beiden Regeln NWD und NRU zu verletzen: Betrachtet man ein System aus der Perspektive des Bell-LaPadula-Modells und findet einen Kanal, der die Regeln nicht verletzt, ist er per Definition erlaubt und damit nicht verdeckt.

2.2.2 Ressourcenkanäle

Ressourcenkanäle (*resource channels*) definieren sich dadurch, dass die an der Kommunikation beteiligten Prozesse gemeinsamen Zugriff auf eine Ressource des Rechners haben und über diesen Zugriff miteinander kommunizieren können.

Typische Ressourcen sind Plattenplatz, Prozessor- (Rechen-) zeit (vgl. das folgende Beispiel), externe Geräte und genutzter Hauptspeicher.

Kommunikation zwischen zwei Prozessen kann beispielsweise stattfinden, indem eine Ressource exklusiv von einem Prozess beansprucht wird: Ein zweiter Prozess, der auf die gleiche Ressource zugreifen will, kann dann feststellen, ob diese bereits gebunden ist oder nicht. Über diese Zustände können Informationsbits übertragen werden.

Das folgende Beispiel zeigt, wie ein Prozess durch Beeinflussung der Systemlast Informationen an einen zweiten Prozess überträgt.

Beispiel

Zwei Prozesse, die auf einem Rechner die aktuelle Systemlast bestimmen können (unter Linux beispielsweise über `/proc/loadavg`), kommunizieren wie folgt miteinander:

Basierend auf der durchschnittlichen Prozessorlast wird ein Grenzwert, der deutlich oberhalb dieses Werts liegt, vereinbart. Um eine „1“ zu übertragen, startet der Senderprozess Berechnungen, die ausreichend aufwendig sind, um die Last über diesen Grenzwert zu treiben; um eine „0“ zu übertragen, unternimmt er nichts.

Der Empfängerprozess untersucht dauernd die Prozessorlast und erkennt ein Muster von Zeiträumen, in denen die Last ober- oder unterhalb des vereinbarten Grenzwerts liegt.

Diese Kommunikation ist störanfällig, wenn andere Benutzer oder das System selbst Prozesse starten und dadurch die Last über den Grenzwert steigt, obwohl der Senderprozess dies nicht veranlasst hat.

Abbildung 2.2 (auf der folgenden Seite) zeigt die Prozessoraktivität über einen Zeitraum von wenigen Minuten – die beiden Blöcke mit 100 % Auslastung wurden erzeugt, indem das Kommando `find /` ausgeführt wurde.

Auch im Beispiel auf Seite 8 wird ein Ressourcenkanal beschrieben: Die gemeinsame Ressource ist das temporäre Verzeichnis `/tmp`, in dem jeder Prozess Dateien erzeugen und dessen Verzeichnisinhalt von jedem Prozess angezeigt werden kann.

2.2.3 Zeitkanäle

Zeitkanäle (*timing channels*) unterscheiden sich durch ein zentrales Kriterium von Ressourcenkanälen: Die beteiligten Prozesse müssen in der Lage sein, zeitliche Abstände zwischen zwei beobachteten Ereignissen zu messen.

Zeitkanäle entstehen, indem der Sender das zeitliche Verhalten eines Teilsystems beeinflusst, beispielsweise das Antwortverhalten eines Servers oder die Ausführungszeit eines Kommandos. Der Empfänger beobachtet diese zeitlichen Änderungen und schließt daraus auf die Nachricht.

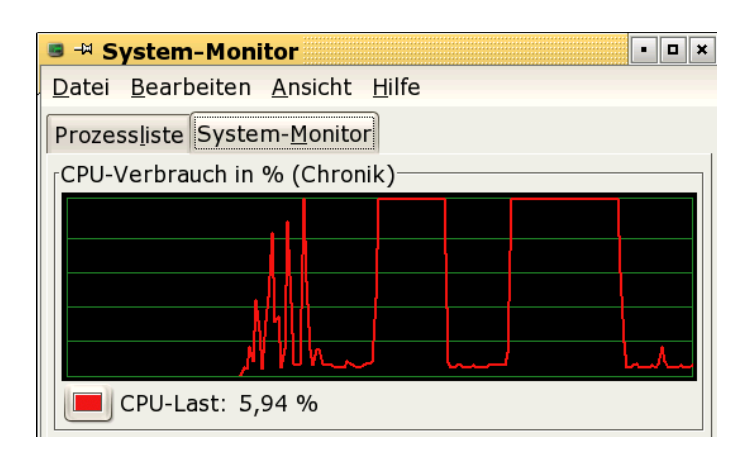


Abbildung 2.2: Die Prozessorbelastung kann durch einen einfachen `find`-Aufruf beeinflusst werden.

Gasser [Gas88] merkt an, dass Zeitkanäle nicht notwendigerweise verschwinden, wenn der Zugriff auf die Uhr des Rechners versperrt wird, da es andere Methoden gibt, Zeitintervalle zu bestimmen.

Zeitkanäle leiden oft unter Rauschen (*noise*): Das Zeitverhalten ist nicht nur vom sendenden Prozess des verdeckten Kanals, sondern auch vom restlichen Systemverhalten abhängig.

2.3 Verwandte Techniken

Dieser Abschnitt beschreibt Techniken, die mit verdeckter Kommunikation verwandt sind: Steganographie und Kryptographie.

2.3.1 Kryptographie

Gegenstand der Kryptographie ist die Verschlüsselung von Daten mit dem Ziel, sie nur autorisierten Anwendern zugänglich zu machen. Unabhängig davon, ob es sich bei den Daten um Texte oder andere, auch binäre, Informationen handelt, wird stets vom „Text“ gesprochen: Der unverschlüsselte, für jedermann lesbare Text wird **Klartext** (*plaintext*) genannt, der verschlüsselte heißt **Chiffretext** (*ciphertext*, vom arabischen Wort „sifr“, leer, Null; englisch „cipher“: Ziffer, Chiffre, Chiffrierung).

Jedes Ver- und Entschlüsselungssystem besteht aus Algorithmen, die den Klartext mit Hilfe eines Schlüssels in die Chiffretext-Version bzw. den Chiffretext und einen Schlüssel in die Klartext-Version zurück wandeln. Es gibt Varianten, die für Ver- und Entschlüsselung den gleichen Schlüssel verwenden (symmetrische Verfahren), und Varianten, die für die beiden Richtungen unterschiedliche Schlüssel benutzen (asymmetrische Verfahren, *Public-Private-Key*-Systeme).

Kryptographie und verdeckte Kanäle lassen sich kombinieren oder jeweils separat verwenden. Während der verdeckte Kanal verschleiert, dass überhaupt Kommunikation stattfindet, macht Kryptographie für alle außer den vorgesehenen Empfängern das Lesen der Nachricht unmöglich (oder schwierig).

2.3.1.1 Symmetrische Verfahren

Bei symmetrischen Verfahren wird ein einziger Schlüssel sowohl für die Ver- als auch für die Entschlüsselung verwendet. Symmetrische Verfahren werden beispielsweise beim Verschlüsseln von Dateisystemen unter Linux oder bei der Verschlüsselung von ZIP-Archiven mit PkZIP [PkZ] eingesetzt.

Typisch sind Block-Chiffren, die nicht einzelne Zeichen, sondern ganze Blöcke verschlüsseln.

Symmetrie definiert sich nicht darüber, dass zum Ver- und Entschlüsseln der gleiche Algorithmus verwendet wird (wie es beispielsweise bei simpler XOR-Verschlüsselung der Fall ist), sondern dass beide Stufen mit demselben Schlüssel arbeiten.

2.3.1.2 Asymmetrische Verfahren

Asymmetrische Verfahren werden meist als *Public-Key*-Verfahren bezeichnet. Sie verwenden ein Schlüsselpaar, das aus

- einem **öffentlichen Schlüssel** (*public key*), der bekannt gemacht wird,
- und einem **privaten Schlüssel** (*private key*), der beim Erzeuger des Schlüsselpaars verbleibt und geheim gehalten werden muss,

besteht. Für die Sicherheit eines solchen Verfahrens ist es wichtig, dass sich der private Schlüssel nicht aus dem öffentlichen berechnen lässt. Dazu werden so genannte Einwegfunktionen verwendet: Schneier [Sch95] definiert eine Funktion f als Einwegfunktion, wenn $f(x)$ für ein gegebenes x einfach zu berechnen ist, sich aber aus einem bekannten Funktionswert $y = f(x)$ nur mit erheblichem Aufwand x (also $f^{-1}(y)$ für eineindeutige Funktionen) berechnen lässt.

Viele Verfahren nutzen Produkte großer Primzahlen: Für zwei große Primzahlen p und q kann man das Produkt pq leicht berechnen – pq wieder zu faktorisieren, erfordert aber viel Rechenzeit.

Bemerkung

Während bei symmetrischen Verfahren die Sicherheit der Übertragung auf guter Wahl und Geheimhaltung der Schlüssel basiert, hängt asymmetrische Verschlüsselung von der Schwierigkeit der Umkehrung der Verschlüsselungsfunktion ab: Wenn ein effizientes Verfahren zur Berechnung der inversen Funktion gefunden wird, macht dies die Verschlüsselungsmethode unbrauchbar. Andererseits erlaubt die asymmetrische Verschlüsselung die Kommunikation zwischen zwei Partnern, ohne dass diese einen vollständigen Schlüssel austauschen müssen, wie es bei symmetrischen Verfahren der Fall ist.

2.3.2 Steganographie

Ziel der Steganographie ist es, Daten verborgen zu übermitteln. Die geheimen Daten werden dabei in ein größeres Datenobjekt eingebettet, das durch diese Veränderung

nur geringfügig gestört wird, so dass die Änderung nicht auffällt. Typische Trägerdaten sind Bilder und Klangdateien: Hier gibt es zahlreiche Implementierungen, von denen weiter unten einige vorgestellt werden.

Ein elementarer Unterschied zwischen Steganographie und verdeckten Kanälen ist die Veränderung der Nutzdaten auf dem regulären Kanal: Im Falle der Steganographie werden die geheimen Daten durch Veränderung der Nutzdaten übertragen. Wer einen alternativen Zugang zur Originaldatei hat, kann also feststellen, dass die Daten manipuliert wurden. Daraus kann noch nicht direkt die geheime Botschaft ermittelt werden (die ja zudem durch den Einsatz von Kryptographie unzugänglich sein kann), aber die Veränderung kann zumindest den Verdacht auslösen, dass geheime Kommunikation mittels Steganographie stattfindet.

Anders ist es bei verdeckten Zeitkanälen: Hier werden die regulär übertragenen Daten nicht modifiziert.

Im Idealfall stört der Prozess des Einbettens nicht die statistischen Eigenschaften der Trägerdaten: Das verhindert dann, dass Analyseverfahren (Steganalysis) die Übertragung mit steganographischen Techniken aufspüren können.

Eine Einführung gibt die für das FBI erstellte Arbeit von Kessler [Kes04].

Eine Liste ungewöhnlicher Steganographie-Anwendungen stellt der „Steganography Wing of the Gallery of CSS Descramblers“ [Tou] zusammen – alle dort beschriebenen Techniken wurden angewandt, um den DeCSS-Code (der zum Lesen von Daten auf Video-DVDs verwendet wird) zu verstecken. Der rechtliche Status der Verbreitung von DeCSS ist unklar, es hat aber einige Urteile gegeben, die DeCSS und auch dessen Verbreitung als legal werten; Gerichte in verschiedenen Ländern beurteilen DeCSS unterschiedlich.

Steganographie ist – genau wie verdeckte Kanäle – eine Variante des allgemeinen Konzepts, Informationen zu verbergen. Eine Übersicht über solche Verfahren gibt der Artikel „Information Hiding—A survey“ [PAK99].

2.3.2.1 Bilder und Klangdateien als Träger

Das Programm `steghide` [Het02] erlaubt die Einbettung von Textdaten in Bilder und Klangdateien. Als Trägerdateien kommen Dateien der Typen JPEG, BMP, WAV und AU in Frage. Beim Einbetten werden statistische Eigenschaften der Farb- bzw. Klangfrequenzen nicht verändert, wodurch das Entdecken der geheimen Daten erschwert wird.

Zusätzlich verschlüsselt `steghide` die eingebetteten Daten über den AES-Algorithmus (*advanced encryption standard*) mit einem 128-Bit-Schlüssel.

Die Anwendung ist einfach; dem Programm werden die Namen von Trägerdatei und einzubettender Datei übergeben:

```
> steghide embed -cf bild.jpg -ef nachricht.txt
Enter passphrase:
Re-Enter passphrase:
embedding "nachricht.txt" in "bild.jpg"... done
```

2.3.2.2 Textdateien als Träger

Das Programm `yescoder` [Yes] versteckt geheime Daten über verschiedene Methoden in ASCII-Dateien, z. B. durch Anhängen von acht Leerzeichen oder einem Tabulatorzeichen an jede Zeile der Trägerdatei.

Tabulatoren werden von den meisten Text-Editoren nicht angezeigt; Abbildung 2.3 zeigt eine kleine Beispieldatei, in der die Zwischenräume zwischen „Test“ und „ABC“ abwechselnd aus vier Leerzeichen und einem Tabulatorzeichen bestehen. In der linken (Standard-) Ansicht des Editors `vi` ist kein Unterschied erkennbar; erst nach Aktivieren der Sonderzeichenanzeige (mit `:set list`) kann man die Tabulatorzeichen (^I) erkennen.

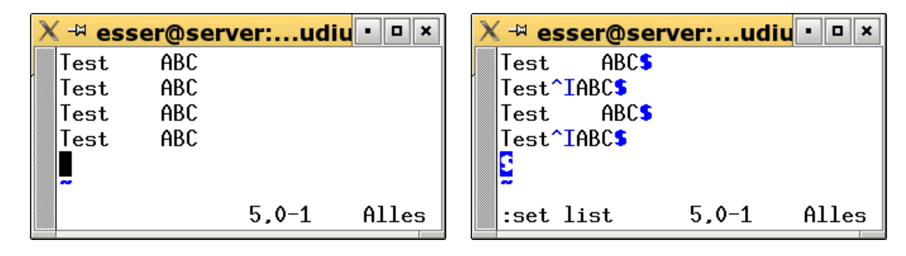


Abbildung 2.3: Text mit Leer- und Tabulatorzeichen; links: Standardansicht, rechts: Ansicht mit Sonderzeichen.

2.4 Anwendungsgebiete für verdeckte Kanäle

In diesem Abschnitt werden einige Szenarien vorgestellt, in denen der Einsatz von verdeckten Kanälen denkbar ist.

2.4.1 Nach Hause telefonieren

Diverse Programme erwarten vor dem ersten Einsatz (oder wenn das Programm über eine Testphase hinaus genutzt werden soll) eine Registrierung beim Hersteller. Üblich ist etwa eine Anmeldung auf der Hersteller-Web-Seite, nach der ein Registrierschlüssel zugeschickt wird, der das Programm freischaltet.

Andere Anwendungen bauen selbständig eine Verbindung zu einem Registrier-Server des Herstellers auf und registrieren sich selbst.

In der Vergangenheit wurden Beispiele dafür entdeckt, dass bei der Registrierung nicht nur die relevanten (und offen abgefragten) Daten, wie Name, Anschrift oder E-Mail-Adresse, sondern auch zusätzliche Daten, wie die Hardware-Ausstattung oder Informationen über auf der Festplatte gespeicherte Dateien, übermittelt wurden. So berichtete zum Beispiel Smith [Smi02] 2002 davon, dass der Windows Media Player (Version 8) beim Einlegen einer Spielfilm-DVD Daten über diese DVD an einen Microsoft-Server überträgt; die Versionen 4.7 und 6 des Browsers Netscape übertragen bei Suchanfragen neben den Suchbegriffen auch weitere Daten über den

Anwender zur Suchmaschine (Online-Artikel von Gerber [Ger02] und Ernst und Luther [EL00]). Im November 2004 erschien auf ZDNet ein Bericht [Ile04], nach dem Lexmarks Windows-Treiber für Drucker und Scanner Hinweise zur Nutzung des Geräts an einen Server des Herstellers übertragen.

Neben Software-Herstellern, die einfach nur sehr neugierig sind und mehr Informationen übermitteln als sie sollten, gibt es auch bösartigere Programme, wie den Virus „Sasser“, die auf der Festplatte nach Kreditkarten- oder anderen persönlichen Informationen suchen und diese dem Programmautor übermitteln; wer beispielsweise in einem Online-Banking-Programm neben Benutzer-ID und PIN auch noch die Transaktionsnummern (TANs) speichert, riskiert bei Virenbefall, dass von seinem Konto aus unautorisierte Überweisungen veranlasst werden.

Netzwerk-*Sniffer*, wie etwa „Analyzer“ [Ana] und „CommView“ [Com], überwachen den Netzwerkverkehr und protokollieren Verbindungen, die von Programmen aufgebaut wurden, und die übertragenen Daten. Hartmann [Har01] beschreibt den Einsatz dieser Tools.

Es ist denkbar, dass Software-Hersteller, die derartige Informationen übertragen wollen, künftig auf verdeckte Kommunikation ausweichen, damit solche Aktivitäten nicht mehr nachgewiesen werden können.

Hier zeigt sich auch das generelle Problem, dass nicht-quelloffene (*Closed-Source-*) Software nicht vertrauenswürdig ist, da mangels Quellcode nicht entschieden werden kann, ob die Software neben ihrem eigentlichen Einsatzzweck noch andere, vom Benutzer unerwünschte, Ziele verfolgt – ein vollständiger Umstieg auf quelloffene (*Open-Source-*) Software löst dieses Problem. (Eine ausführliche Beschreibung der unterschiedlichen Bedeutung von *Open Source* vs. *Closed Source* und „freier Software“ vs. „proprietärer Software“ gibt Stallman [Sta02].)

2.4.2 Betriebssystem-Scanner

Über das Antwortverhalten eines Rechners beim Zusenden bestimmter TCP-Pakete lässt sich eine Signatur erstellen und mit einer Signaturdatenbank vergleichen – daraus kann man erkennen, welches Betriebssystem läuft. Spangler [Spa03] stellt drei derartige Verfahren vor:

- Das Unix-Kommando `nmap` [Nma] führt einen *Port-Scan* aus. Findet es drei für seine Zwecke nutzbare Ports (offener TCP-Port, geschlossener TCP-Port, geschlossener UDP-Port), schickt es TCP- und UDP-Pakete an diese Ports und analysiert das Antwortverhalten. Anhand einer Datenbank mit den Reaktionen bekannter Betriebssysteme wird der statistisch ähnlichste Eintrag gesucht und als Antwort zurückgegeben.
- Bei SYN_RCVD (RINGv2) wird zunächst ein *Port-Scan* des Zielrechners durchgeführt, da ein offener Port benötigt wird. Dann wird eine lokale Firewall-Regel erzeugt, die Pakete vom Zielrechner blockiert. Der Zielrechner wird auf dem offenen Port kontaktiert, aber das Antwortpaket kann nicht zugestellt werden. Der Zielrechner wird mehrere weitere Zustellversuche unternehmen,

das Programm misst die Verzögerungen zwischen den erneuten Übertragungen. Auch hieraus wird eine Signatur erstellt, die mit den Einträgen in einer Signaturdatenbank verglichen wird.

- Xprobe2 [Xpr] verwendet eine Methode, die die Software-Entwickler als *fuzzy matching* bezeichnen: Mehrere Tests werden unabhängig durchgeführt und die Testergebnisse mit der Signaturdatenbank verglichen; dabei sind jeweils vier Ergebniswerte möglich, mit denen die Wahrscheinlichkeit für eine Übereinstimmung ausgedrückt wird. Die Werte aller unabhängigen Tests werden aufaddiert, so dass sich eine Rangreihenfolge unter den der Signaturdatenbank bekannten Betriebssystemen ergibt – das System mit dem höchsten Übereinstimmungswert ist dann mit großer Wahrscheinlichkeit ein Treffer.

Dies ist ein Beispiel für verdeckte Systemkanäle: Es gibt keinen Kommunikationspartner, der aktiv an der verdeckten Kommunikation teilnimmt; die Scan-Tools nutzen einfach Standardfunktionen des Betriebssystems, um ihre Erkenntnisse zu gewinnen.

2.5 Kapazität verdeckter Kanäle: Informationstheorie

Dieser Abschnitt führt in die Grundlagen der Informationstheorie ein, so weit sie für diese Arbeit relevant sind. Insbesondere werden Kanäle und Kapazität behandelt.

Informationstheorie beschreibt die mathematischen Grundlagen der Kommunikation und Datenübertragung. Sie basiert auf der 1948 veröffentlichten Arbeit „A Mathematical Theory of Communication“ von Shannon [Sha48].

MacKay [Mac03] nennt eine der Hauptanwendungen der Informationstheorie:

> Information theory and coding theory offer an alternative (and much more exciting) approach: we accept the given noisy channel as it is and add communication systems to it so that we can detect and correct the errors introduced by the channel.

Fehlererkennung und -korrektur helfen, einen gestörten Kanal nutzbar zu machen.

2.5.1 Wahrscheinlichkeitsrechnung

In diesem Abschnitt werden einige grundlegenden Konzepte der Wahrscheinlichkeitsrechnung beschrieben, die für das Verständnis dieser Arbeit notwendig sind.

Es wird überwiegend die Notation aus den Büchern von Mathar [Mat96] und Mathar/Pfeifer [MP90] verwendet.

Ist $\Omega \neq \varnothing$ eine Grundmenge und $\mathcal{A} \subseteq \text{Pot}(\Omega)$ eine Menge von Teilmengen der Grundmenge, dann heißt $\mathcal{A}$ **σ-Algebra** über Ω, falls 1) Ω ein Element von $\mathcal{A}$ ist, 2)

Definition

mit jeder Menge $A \in \mathcal{A}$ auch ihr Komplement $\overline{A}$ zu $\mathcal{A}$ gehört und 3) aufzählbare Vereinigungen von Elementen von $\mathcal{A}$ ebenfalls zu $\mathcal{A}$ gehören.

Aus jeder Teilmenge $\mathcal{E}$ von Pot(Ω) kann man eine σ-Algebra erzeugen, indem man den Durchschnitt aller σ-Algebren bildet, die $\mathcal{E}$ enthalten:

$$\sigma(\mathcal{E}) := \bigcap \{\mathcal{A} | \mathcal{A} \text{ ist } \sigma\text{-Algebra über } \Omega \text{ und } \mathcal{E} \subseteq \mathcal{A}\}$$

Für $\Omega = \mathbb{R}$ und die Menge $\mathcal{E}$ aller links offenen und rechts abgeschlossenen Intervalle $(a, b]$ heißt $\mathcal{B}^1 := \sigma(\mathcal{E})$ **Borel-σ-Algebra**.

Ergänzt man eine σ-Algebra $\mathcal{A}$ über Ω noch um eine Abbildung $P : \mathcal{A} \to [0, 1]$ mit $P(\Omega) = 1$ und $P(\cup_{n \in \mathbb{N}} A_n) = \sum_{n \in \mathbb{N}} P(A_n)$ für paarweise disjunkte A_i ($i \neq j \Rightarrow A_i \cap A_j = \emptyset$), dann heißt das Tripel $(\Omega, \mathcal{A}, P)$ **Wahrscheinlichkeitsraum**, und P heißt **Wahrscheinlichkeitsmaß** auf $\mathcal{A}$.

Ist $X : \Omega \to \mathbb{R}$ eine Abbildung mit

$$\{X \in B\} := X^{-1}(B) = \{\omega \in \Omega | X(\omega) \in B\} \in \mathcal{A} \text{ für alle } B \in \mathcal{B}^1$$

dann heißt X **Zufallsvariable** auf Ω, und X heißt **messbar**.

Im Folgenden werden nur noch endliche Grundmengen Ω betrachtet.

Ein **Zufallsvektor** ist ein n-Tupel $\mathbf{x} = (x_1, \ldots, x_n)$ mit folgenden Eigenschaften:

1. $x_i \geq 0, i = 1, \ldots, n$
2. $\sum_{i=1}^{n} x_i = 1$

Ein Zufallsvektor lässt sich als Beschreibung der Wahrscheinlichkeiten von n disjunkten Zufallsereignissen betrachten, von denen stets genau eines eintreten muss.

Der Zufallsvektor $\mathbf{x}$ entspricht der Kombination aus Wahrscheinlichkeitsraum $(\Omega, \mathcal{A}, P)$ mit $\Omega = \{a_1, ..., a_n\} \subset \mathbb{R}$ und $\mathcal{A} = \text{Pot}(\Omega)$ sowie der Zufallsvariablen $X : \Omega \to \mathbb{R}, x \mapsto x$ mit

$$P(X = a_i) = P(X^{-1}(\{a_i\})) = P(\{a_i\}) = x_i \quad (i = 1, \ldots, n)$$

Im Folgenden heißt die Grundmenge $\Omega = \{a_1, \ldots, a_n\}$ auch **Alphabet**, und statt $P(X = a_i)$ oder $P(\{a_i\})$ wird kurz $P(a_i)$ geschrieben.

Beispiel

Als einfaches Beispiel sei für einen fairen Würfel (mit Wahrscheinlichkeit $1/6$ für jede Seite) E_i das Zufallsereignis „Beim Werfen liegt Seite i oben“. Setzt man dann $p_i := P(E_i) = 1/6$, ist $(p_1, \ldots, p_6)$ ein Zufallsvektor.

2.5.2 Informationsgehalt, Entropie

Es folgen einige grundlegende Definitionen. Als Konvention wird festgelegt, dass mit $\log$ stets der duale Logarithmus $\log_2$ gemeint ist. Der duale Logarithmus lässt sich mit Hilfe des natürlichen Logarithmus' $\ln = \log_e$ über

$$\log(x) = \frac{\ln(x)}{\ln 2}$$

berechnen, denn allgemein gilt:

$$\log_b(x) = \frac{\ln(x)}{\ln(b)}$$

Es wird nun der Begriff „Informationsgehalt“ definiert. Sinn der Definition ist, aus den Wahrscheinlichkeiten für das Auftreten einzelner Zeichen eine Bewertung für eine empfangene Zeichenkette abzuleiten. Dazu sollten die folgenden Eigenschaften erfüllt sein:

- Je unwahrscheinlicher das Auftreten eines Zeichens ist, desto größer soll der Informationsgehalt sein. (Das Auftreten seltener Zeichen entspricht nicht der Erwartung, hat Ausnahmecharakter.)
- Ein Zeichen, das mit Wahrscheinlichkeit 1 auftritt, hat den Informationsgehalt 0. (In diesem Fall gibt es keinen Zugewinn an Information, nachdem das Zeichen empfangen wurde – wegen der Sicherheit war das ja im Voraus bekannt.)
- Der Informationsgehalt einer Zeichenkette ist die Summe der Informationsgehalte der einzelnen Zeichen.

Seien Ω, X, P wie oben, $P(a_i) = p_i$. Dann ist für jedes i

Definition

$$h(a_i) := \log \frac{1}{P(a_i)} = \log \frac{1}{p_i} \tag{2.1}$$

der **(Shannon-) Informationsgehalt** von a_i.

Abbildung 2.4 zeigt den Informationsgehalt $h(p)$ für $0 \leq p \leq 1$.

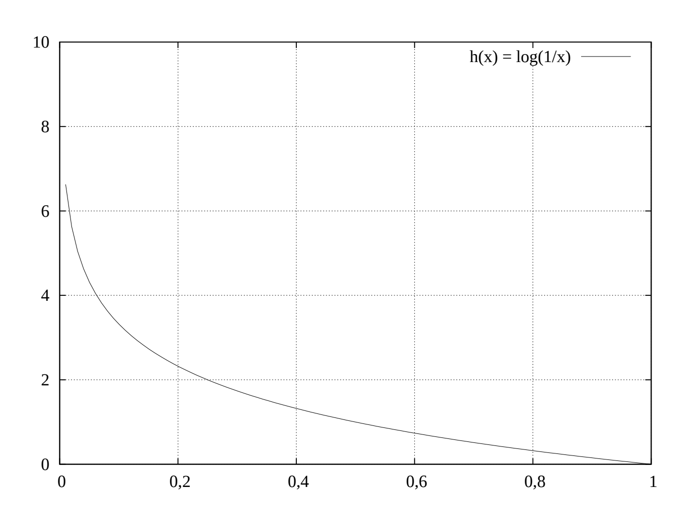


Abbildung 2.4: Informationsgehalt h.

Summiert man alle Informationsgehalte, jeweils multipliziert mit den Wahrscheinlichkeiten, erhält man die Entropie – sie ist ein Maß für den Informationsgewinn, den man vom Ausgang eines Experimentes erwarten kann: Je größer die Entropie ist, desto höher ist die Zufälligkeit eines Experiments; wenn die Entropie 0 ist, ist der Ausgang bereits vor dem Experiment festgelegt.

Definition

Die **Entropie** ist der (gewichtete) durchschnittliche Informationsgehalt:

$$H(X) := \sum_{i=1}^{n} P(a_i) \log \frac{1}{P(a_i)} = \sum_{i=1}^{n} p_i \log \frac{1}{p_i} = \sum_{i=1}^{n} p_i h(a_i) \tag{2.2}$$

wobei für $p = 0$ die Nulldivision $p \log \frac{1}{p} := 0$ gesetzt wird.

Die folgenden Beispiele betrachten die Entropie von drei Szenarien: dem sicheren Ereignis, der 50:50-Chance und ungleichen Chancen (bei zwei möglichen Ausgängen).

Beispiel

1. Bei einer 1-elementigen Ergebnismenge ist die Wahrscheinlichkeit für dieses eine Ereignis 1 – die Entropie ist 0:

$$\sum P(x) \log \frac{1}{P(x)} = 1 \times \log \frac{1}{1} = 0$$

2. Beim Münzwurf (mit einer fairen Münze) haben die beiden möglichen Ereignisse (Kopf und Zahl) beide die Wahrscheinlichkeit 0,5. Die Entropie ist 1:

$$\sum P(x) \log \frac{1}{P(x)} = 0,5 \times \log \frac{1}{0,5} + 0,5 \times \log \frac{1}{0,5} = \log 2 = 1$$

   Damit sind die beiden Zufallsvektoren (1) (sicheres Ereignis) und $(\frac{1}{2}, \frac{1}{2})$ (fairer Münzwurf) normierend für die Entropie.

3. Beim Wurf einer unfairen Münze erscheint mit Wahrscheinlichkeit 0,9 Kopf und mit Wahrscheinlichkeit 0,1 Zahl. Daraus ergibt sich für die Entropie:

$$\sum P(x) \log \frac{1}{P(x)} = 0,9 \times \log \frac{1}{0,9} + 0,1 \times \log \frac{1}{0,1} \approx 0,469$$

   Für Kopfwahrscheinlichkeiten 0,99, 0,999, 0,9999 und 0,99999 ergeben sich die auf vier Nachkommastellen gerundeten Entropiewerte 0,0808, 0,0114, 0,0015 und 0,0002. Die Entropie immer wahrscheinlicher werdender eindeutiger Ausgänge strebt also gegen 0 – das ist die Entropie des sicheren Ereignisses mit Wahrscheinlichkeit 1.

Abbildung 2.5 zeigt die Entropie für zweielementige Zufallsvektoren $(p, 1-p)$: Die Funktion ist symmetrisch, da die Entropie nicht von der Reihenfolge der Wahrscheinlichkeiten im Zufallsvektor abhängt.

Die Entropie ist ein Maß dafür, wie viele Bits man bei einer binären Kodierung (mit variabler Wortlänge) im Schnitt benötigt. Solche Kodierungen sind wie folgt definiert:

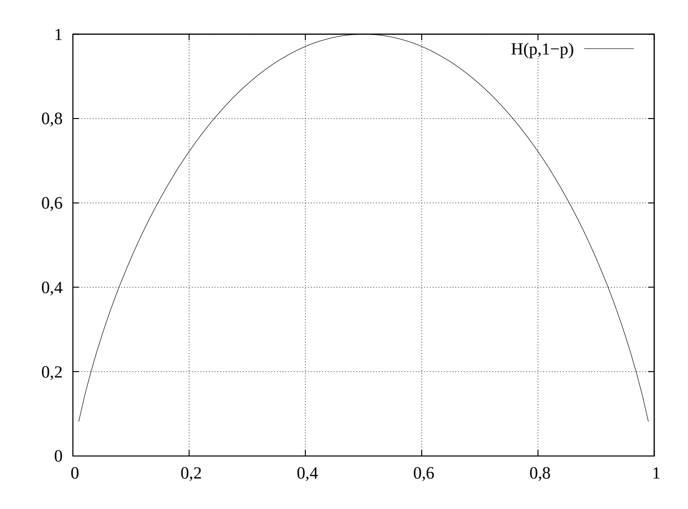


Abbildung 2.5: Entropie $H(p, 1-p)$.

Definition

Ist $B = \{0, 1\}$ das binäre Alphabet und B^* die Menge aller Worte über B (einschließlich des leeren Worts ε), dann heißt eine Abbildung $c : \Omega \to B^*$ **binäre Kodierung**, falls sie **präfixfrei** ist – d. h., kein $c(a_i)$ ist Präfix eines anderen $c(a_j)$. Für $a_i \in \Omega$ ist $|c(a_i)|$ die Wortlänge (des kodierten Worts). Sind alle $|c(a_i)|$ gleich, heißt C **binäre Kodierung mit fester Wortlänge**, anderenfalls **binäre Kodierung mit variabler Wortlänge**. Für ein Wort $w = x_1x_2 \dots x_n \in \Omega^*$ ist $c(w)$ die Konkatenation $c(x_1)c(x_2) \dots c(x_n)$.

Es folgen weitere informationstheoretische Begriffsdefinitionen.

Definition

Sind $\mathbf{x} = (p_1, p_2, \dots, p_m)$ und $\mathbf{y} = (q_1, q_2, \dots, q_n)$ zwei Zufallsvektoren mit $p_i = P(X = a_i)$ und $q_j = P(Y = a_j)$, dann heißt

$$H(X, Y) := \sum_{i=1}^{m} \sum_{j=1}^{n} P(X = a_i, Y = b_j) \frac{1}{\log P(X = a_i, Y = b_j)} \tag{2.3}$$

(gemeinsame) Entropie von (X, Y).

$$H(X|Y = b_j) := \sum_{i=1}^{m} P(X = a_i|Y = b_j) \frac{1}{\log P(X = a_i|Y = b_j)} \tag{2.4}$$

heißt **bedingte Entropie von X, gegeben** $\mathbf{Y = b_j}$.

$$H(X|Y) := \sum_{j=1}^{n} P(Y = b_j) H(X|Y = b_j) \tag{2.5}$$

ist die **bedingte Entropie von X unter Y**.

$$I(X, Y) := H(X) - H(X|Y) \tag{2.6}$$

heißt **Transinformation** von X und Y.

Satz

Satz 2.1 (Additivität und Symmetrie) Mit X und Y wie oben gelten:

1. $H(X, Y) = H(X) + H(Y|X) = H(Y, X)$

2. $$\begin{aligned} I(X,Y) &= H(X) - H(X|Y) = H(Y) - H(Y|X) \\ &= H(X) + H(Y) + H(X,Y) = I(Y,X) \end{aligned}$$

3. Sind X und Y stochastisch unabhängig (d. h., $P(X = a_i, Y = b_j) = P(X = a_i)P(Y = b_j)$ für alle i, j), dann gilt $H(X,Y) = H(X) + H(Y)$. □

2.5.3 Mathematisches Kanalmodell

Die folgende Definition (die mit leicht abgewandelter Notation aus dem Buch von MacKay [Mac03] übernommen wurde) führt ein mathematisches Kanalmodell ein – vorausgesetzt wird hier, dass die auf dem Kanal versendeten einzelnen Zeichen stochastisch unabhängig sind und auch das empfangene Zeichen nicht von der bisherigen Übertragung abhängt.

Definition

Ein **diskreter gedächtnisloser Kanal** Q besteht aus einem Eingabealphabet Ω_X, einem Ausgabealphabet Ω_Y und einer Menge von Wahrscheinlichkeitsverteilungen $P_x(y)$ (für jedes $x \in \Omega_X$). Statt $P_x(y)$ wird auch $P(y \mid x)$ geschrieben.

Ein diskreter gedächtnisloser Kanal modelliert einen gestörten Kanal: $P(y \mid x)$ ist die Wahrscheinlichkeit, dass beim Versand des Zeichens x das Zeichen y empfangen wird.

Sind X und Y die zugehörigen Zufallsvariablen, ist $P(y \mid x)$ eine Kurzschreibweise für $P(Y = y \mid X = x)$, und man kann die Transinformation $I(X,Y) = H(X) - H(X|Y)$ betrachten: Sie gibt an, wie sich die Unsicherheit über ein versandtes x (die Entropie $H(X)$) durch Kenntnis des empfangenen y verringert.

Für jede feste Verteilung von X (die sich durch einen Zufallsvektor $\mathbf{p} = (p_1, \ldots, p_n)$ angeben lässt) kann man $I(X,Y)$ betrachten. Für dieses feste $\mathbf{p}$ misst $I(X,Y)$, wie viel Information der Empfang eines Zeichens über das versandte Zeichen verrät.

Maximiert man $I(X,Y)$ über alle Input-Verteilungen, erhält man ein Maß dafür, wie viel Information der Kanal fehlerfrei übertragen kann:

Definition

Die **Kapazität** eines diskreten gedächtnislosen Kanals Q ist das Maximum der Transinformation über alle Input-Verteilungen:

$$C(Q) := \max_{\mathbf{p}} I(X,Y)$$

Die Kapazität ist ein Wert zwischen 0 und 1 und beschreibt die prozentuale Nutzbarkeit eines Kanals: Überträgt ein Kanal formal 100 Bit pro Sekunde, hat aber nur eine Kapazität $C = 0{,}8$, dann können über ihn maximal 80 Bit pro Sekunde fehlerfrei übertragen werden. (Um die Kapazität annähernd zu erreichen, müssen die Datenbits geeignet – mit Fehlerkorrekturverfahren – in die formalen 100 Bit kodiert werden.)

Shannon hat bewiesen, dass man sich der theoretischen Kapazität beliebig nähern und gleichzeitig die Fehlerrate beliebig klein halten kann. Das ist die Aussage des **Shannonschen Fundamentalsatzes** (**Hauptsatz der Informationstheorie**), weitere Informationen bietet das Buch von Mathar [Mat96].

2.5.4 Informationstheorie und verdeckte Kanäle

Moskowitz und Miller [MM94] haben ungestörte verdeckte Zeitkanäle (*Simple timing channels*) untersucht und die Kapazität eines Kanals, auf dem k Zeichen $s_1, s_2, \ldots, s_k$ mit Übertragungszeiten $t_1, t_2, \ldots, t_k$ versandt werden, als

$$C = \log \omega$$

bestimmt, wobei $\omega > 1$ die positive Nullstelle von

$$1 - \sum_{i=1}^{k} x^{-t_i}$$

ist.

Dabei sind die t_i ganzzahlig ($t_i \in \mathbb{N}$) und zählen Zeitintervalle, d. h., jedes t_i lässt sich über einen Intervallfaktor f in Sekunden umrechnen. Für spätere Betrachtungen wird der Fall mit $k = 2$ und $t_1 = 1$ untersucht, so dass sich der Ausdruck zu

$$1 - x^{-1} - x^{-t_2}$$

vereinfacht.

Sei also $t_1 = 1$. Setzt man $t := t_2 - 1$ (und betrachtet t_2 als Verzögerung von t_1 um t), dann kann man $0 = 1 - x^{-t_1} - x^{-t_2} = 1 - x^{-1} - x^{-t-1}$ umformen in $x^{t+1} - x^t - 1 = 0$.

Moskowitz und Miller nennen den so bestimmten Kanal $T(1, 1+t)$. Die berechnete Kapazität misst die Anzahl der Bits, die pro Zeitintervall übertragen werden können.

Für diese Arbeit wird nur $T(1, 1+t)$ betrachtet, weil im weiteren Verlauf ein verdeckter Kanal generiert wird, der diese Parameter verwendet: Die unverzögerte Übertragung eines 4-KByte-Blocks vom Web-Server zum Client benötigt im Schnitt $d \approx$ 0,0328 s. Wird nun für einige Datenpakete eine Verzögerung eingeführt, die annähernd ein Vielfaches td von d ist, und normiert man die Zeiteinheit auf d, beschreibt $T(1, 1+t)$ den daraus resultierenden Kanal, der unverzögert Nullen und verzögert Einsen sendet.

Das folgende Beispiel sendet Einsen durch neunfache Verzögerung (also zehnfache Übertragungszeit im verzögerten Fall):

Beispiel

Für $t = 9$ hat die Gleichung die Form $x^{10} - x^9 - 1 = 0$. Abbildung 2.6 (auf der folgenden Seite) zeigt den Funktionsgraph, eine Nullstellensuche mit Octave [Oct, Eat97] ergibt als positive Nullstelle den Wert $\approx 1,19749$. (Das Polynom hat zehn Nullstellen, davon acht imaginäre, eine negative und eine positive.) Damit ist die Kapazität $C_{1,10}$ in erster Näherung $\log 1,19749 \approx 0,26$.

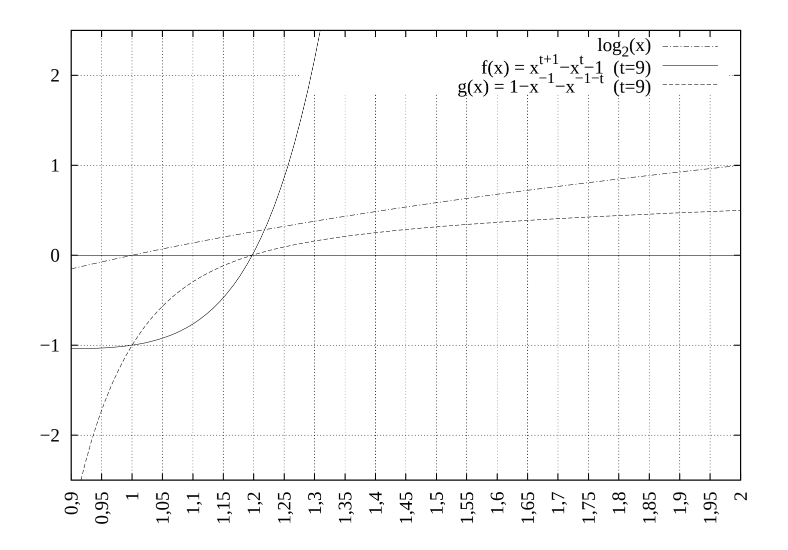


Abbildung 2.6: Nullstellensuche in $x^{10} - x^9 - 1$.

2.6 $\mathbb{B}$-Vektorräume und lineare Algebra

Für das anschließende Kapitel über Fehlerkorrektur werden einige Informationen über endliche Vektoren aus Nullen und Einsen benötigt. Die Menge aller solcher Vektoren (fester Länge n) bildet einen Vektorraum, wie die folgenden Beschreibungen zeigen.

Definiert man auf $\mathbb{B} := \{0, 1\}$ die binäre Addition und Multiplikation durch

+	0	1
0	0	1
1	1	0

*	0	1
0	0	0
1	0	1

dann ist $(\mathbb{B}, +, *)$ ein **Körper**. Über jedem Körper K kann man **Vektorräume** definieren: Das sind im Allgemeinen Mengen V, auf denen eine Addition $+$ und eine Skalarmultiplikation kv (für $k \in K$ und $v \in V$) definiert sind, die gemäß der Definition aus dem Buch von Lang [Lan70] folgenden Regeln gehorchen:

$\forall u, v, w \in V, k, l \in K :$

(1)	$(u + v) + w = u + (v + w)$	Addition assoziativ
(2)	$\exists O \in V : O + v = v, v + 0 = v$	Nullvektor
(3)	$\exists - v \in V : v + (-v) = O$	Inverses bzgl. Addition
(4)	$u + v = v + u$	Addition kommutativ
(5)	$k(u + v) = ku + kv$	distributiv (i)
(6)	$(k + l)v = kv + lv$	distributiv (ii)
(7)	$(kl)v = k(lv)$	Skalarmult. assoziativ
(8)	$1v = v$ (für $1 \in K$)	Multiplikation mit 1

Für den Sonderfall $K = \mathbb{B}$ lassen sich einige der Regeln anders ausdrücken bzw. fallen weg:

$(3')$	$v + v = 0$	Selbstinvers bzgl. Addition
$(5')$	$0v = O$	Multiplikation mit 0
(6)	–	(entfällt)
(7)	–	(entfällt)

Die Regeln (1), (2), $(3')$, (4), $(5')$ und (8) sind für $\mathbb{B}$ äquivalent zu (1)–(8), denn es gilt Folgendes:

Zu (3): Für $-v := v$ gilt $v + (-v) = O$ nach $(3')$.

Zu (5): Für $k = 0$: $0(u + v) = O$ (nach $(5')$), und $0u + 0v = O + O = O$. Für $k = 1$: $1(u + v) = u + v = 1u + 1v$.

Zu (6): $(0+0)v = 0v = O = O+O = 0v+0v$, $(0+1)v = 1v = v = O+v = 0v+1v$, $(1 + 0)v$ analog, $(1 + 1)v = 0v = O = v + v = 1v + 1v$.

Zu (7): $(0*0)v = 0v = O = 0(0v)$, $(0*1)v = 0v = 0(1v)$, $(1*0)v = 0v = O = 1O = 1(0v)$, $(1 * 1)v = 1v = 1(1v)$.

Anschaulich besteht in $\mathbb{B}$ (und ebenso in $\mathbb{B}$-Vektorräumen) kein Unterschied zwischen Addition und Subtraktion, da $1 = -1$.

Im Rest dieser Arbeit werden nur die n-dimensionalen $\mathbb{B}$-Vektorräume verwendet, die aus n-stelligen Vektoren mit Elementen aus $\mathbb{B}$ bestehen:

$$\mathbb{B}^n := \{(b_1, \ldots, b_n) \quad | \quad b_i \in \mathbb{B} \text{ für } i = 1, \ldots, n\}$$

Definition

Für Skalare $k_1, \ldots, k_m$ und Vektoren $v_1, \ldots, v_m$ heißt

$$\sum_{i=1}^{m} k_i v_i = k_1 v_1 + \cdots + k_m v_m$$

eine **Linearkombination** der v_i.

Eine Folge $(v_1, \ldots, v_m)$ heißt **linear unabhängig**, wenn sich jede Linearkombination v der v_i nur auf genau eine Weise aus den v_i linear kombinieren lässt. Sie heißt **Basis** des Vektorraums V, wenn sie linear unabhängig ist und jedes Element aus V als Linearkombination der Vektoren darstellbar ist. Jede Basis für $\mathbb{B}^n$ besteht aus n Vektoren, und n heißt **Dimension** des Vektorraums. Die **Standardbasis** von $\mathbb{B}^n$ ist $\{b_i | 1 \leq i \leq n\}$, wobei der Vektor b_i aus lauter Nullen und einer Eins an Position i besteht: $(b_i)_j = 1$ für $i = j$, $(b_i)_j = 0$ für $i \neq j$. Zum Beispiel ist $\{(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)\}$ die Standardbasis von $\mathbb{B}^4$.

Eine Teilmenge W eines Vektorraums V heißt **Unterraum**, wenn W unter Vektoraddition und Skalarmultiplikation abgeschlossen ist, d. h.: Für alle $v, w \in W$ und $k \in \mathbb{B}$ liegen auch $v + w$ und kv in W.

Eine Abbildung $f : \mathbb{B}^m \to \mathbb{B}^n$ heißt **linear**, wenn $f(av + bw) = af(v) + bf(w)$ für alle $a, b \in \mathbb{B}$ und $v, w \in \mathbb{B}^m$ gilt. Die Wertemenge von f, also $\{f(v) | v \in \mathbb{B}^m\} \subseteq \mathbb{B}^n$,

wird mit Bild(f) bezeichnet. Bild(f) ist ein Unterraum von $\mathbb{B}^n$, seine Dimension heißt **Rang** von f.

Die folgenden zwei Beispiele sind häufig verwendete lineare Abbildungen zwischen zwei Vektorräumen unterschiedlicher Dimension:

Beispiel

Die Abbildung $p : \mathbb{B}^3 \rightarrow \mathbb{B}^2$, definiert durch $p((x, y, z)) := (x, y)$, ist eine lineare Abbildung. Bild$(p) = \mathbb{B}^2$, und der Rang von p ist 2. p heißt **Projektion**.

Beispiel

Die Abbildung $i : \mathbb{B}^2 \rightarrow \mathbb{B}^3$, definiert durch $i((x, y)) := (x, y, 0)$, ist eine lineare Abbildung. Bild(i) ist eine echte Teilmenge von $\mathbb{B}^3$, und der Rang von i ist 2. i heißt **Einbettung**.

Jede lineare Abbildung $f : \mathbb{B}^m \rightarrow \mathbb{B}^n$ kann als $m \times n$-Matrix F (mit n Spalten und m Zeilen) dargestellt werden, dabei ist $F_{ij} = (f(b_i))_j$ für Basisvektoren (b_i). Die Abbildungen p und i aus den obigen Beispielen haben die Matrixdarstellungen P und I:

$$P = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}, \qquad I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Angewendet werden diese Matrizen durch Rechtsmultiplikation; im Beispiel gelten $p(x) = xP$ und $i(x) = xI$.

In der allgemeinen Terminologie der linearen Algebra sind solche Matrizen die Darstellungen linearer Abbildungen bezüglich der Standardbasen von $\mathbb{B}^m$ und $\mathbb{B}^n$ – wählt man andere Basen, erhält man andere Matrizen.

2.7 Fehlererkennung und -korrektur

Verfahren zur Fehlererkennung und -korrektur werden benutzt, wenn Kanäle fehlerbehaftet sind, d. h., wenn die Wahrscheinlichkeit für die korrekte Übertragung eines Zeichens kleiner 1 ist.

Die Integration einer Fehlererkennung (z. B. in Form von einfachen Prüfsummen) ist ausreichend, wenn es einen Rückkanal gibt, über den ein fehlerhaftes Datenpaket neu angefordert werden kann. Fehlt ein solcher Rückkanal, empfiehlt sich ein Fehlerkorrekturverfahren, welches das Wissen über eine gewisse Störung des Kanals in die Kodierung einbezieht: Über Redundanz kann der Empfänger dann auch aus einer gestörten Übertragung die korrekte Mitteilung dekodieren, solange die Fehlerquote auf den angenommenen Rahmen begrenzt bleibt.

Sowohl Fehlererkennung als auch -korrektur verringern die Kapazität eines Kanals, da neben den eigentlichen Daten zusätzliche Informationen übertragen werden müssen.

Für die Implementierung des verdeckten Kanals, die im Rahmen dieser Diplomarbeit erfolgte, war der Einsatz von Fehlererkennung nicht hilfreich, da der oben erwähnte Rückkanal fehlt: Die Kommunikation verläuft nur einseitig vom Apache-Server zum HTTP-Proxy.

2.7.1 Prüfsummen

Mit Hilfe von Prüfsummen kann der Empfänger einer Botschaft versuchen zu prüfen, ob die Übertragung fehlerfrei war. Die einfachste Möglichkeit, m Bits mit einer Prüfsumme zu versehen, ist die Addition aller Bits modulo 2, also die Addition in $\mathbb{B}$:

$$c := c((b_1, \ldots, b_m)) := \sum_{i=1}^{m} b_i \qquad (+ = +_{\mathbb{B}})$$

Wird nun statt $(b_1, \ldots, b_m)$ der ein Bit längere Block $(b_1, \ldots, b_m, c)$ übertragen, berechnet der Empfänger die Summe aller Bits, die 0 sein muss. Tritt bei der Übertragung ein einzelner Bitfehler auf, ist die Prüfsumme 1, und es wird ein Fehler erkannt. Treten jedoch $2n$ Bitfehler ($n \in \mathbb{N}, n > 0$) auf, ist die Prüfsumme ebenfalls 0, und die Fehler bleiben unerkannt.

Dem Problem mehrfacher Fehler kann man begegnen, indem mehrere Prüfsummen berechnet werden, wobei jede der Prüfsummen nur eine Auswahl der Bits aufaddiert. Bei günstiger Auswahl der Prüfsummenfunktionen ist sogar eine Fehlerkorrektur möglich, d. h., es werden nicht nur Fehler erkannt, sondern der Empfänger kann aus der fehlerhaften Nachricht die ursprüngliche Nachricht rekonstruieren.

Prüfsummen führen Redundanz ein und erhöhen damit die zu übertragende Datenmenge bzw. verringern die Kanalkapazität.

Ein systematischer Ansatz, der mit mehreren Prüfsummen arbeitet, ist von Hamming und wird im folgenden Unterkapitel behandelt.

2.7.2 Hamming

In einer Veröffentlichung aus dem Jahr 1950 [Ham50] beschreibt Hamming ein Fehlerkorrekturverfahren, das in der Lage ist, einen Fehler pro übertragenem Block selbständig zu korrigieren.

Hamming verwendet eine Blockkodierung, die Blöcke aus m Bits mit Hilfe von k zusätzlichen Paritätsbits in Blöcke der Länge $n = m + k$ wandelt.

Die Fehlerkorrektur nutzt eine geometrische Interpretation der n-dimensionalen Punktmenge $\{0, 1\}^n$: Über die so genannte Hamming-Distanz

$$D((a_1, \ldots, a_n), (b_1, \ldots, b_n)) := \sum_{i=1}^{n} |a_i - b_i|$$

wird eine Metrik auf $\{0, 1\}^n$ mit den üblichen Eigenschaften definiert: $\forall x, y, z \in \{0, 1\}^n$:

$$D(x, y) = D(y, x); \qquad D(x, y) = 0 \iff x = y$$

$$D(x, y) + D(y, z) \geq D(x, z) \quad \text{(Dreiecksungleichung)}$$

Bemerkung

Die Hamming-Distanz lässt sich noch auf eine zweite, äquivalente Weise definieren: Es gilt

$$D((a_1, \ldots, a_n), (b_1, \ldots, b_n)) = |\{i : a_i \neq b_i\}|$$

In dieser Gleichung wird D als Anzahl der abweichenden Punkte zweier $\{0,1\}$-Vektoren beschrieben, die oben stehende Definition ist die Summe der Abstände der einzelnen „Koordinaten" und erlaubt eine geometrische Veranschaulichung, wenn man $\{0,1\}^n$ als Teilmenge von $\mathbb{R}^n$ auffasst.

Mit $m < n$ ist klar, dass jede Kodierfunktion $f : \{0,1\}^m \to \{0,1\}^n$ nicht-surjektiv ist, also Bild(f) eine echte Teilmenge von $\{0,1\}^n$ ist: $2^n - 2^m$ Punkte werden nicht getroffen.

Um Fehlerkorrektur zu ermöglichen, wählt man nun f so, dass je zwei Bildwerte $f(x_1)$ und $f(x_2)$ mindestens Hamming-Distanz 3 haben. Kippt durch den Übertragungsfehler eines $f(x)$ eines der Bits, hat der falsch empfangene Wert die Hamming-Distanz 1 vom richtigen Funktionswert, aber mindestens Hamming-Distanz 2 von allen „falschen" Werten – das korrekte Ergebnis muss also jener Wert aus Bild(f) sein, der minimale Hamming-Distanz zum falschen Wert hat.

Kippen zwei oder mehr Bits bei der Übertragung, versagt diese Fehlerkorrektur. Es ist also wichtig, die Größe eines Blocks zu begrenzen, um die Wahrscheinlichkeit mehrerer Fehler innerhalb eines Blocks zu verringern. In der Implementierung wurde die Blockgröße 6 Bit gewählt, da (eine limitierte Auswahl von) ASCII-Zeichen bei der Übertragung in 6 Bit kodiert werden.

2.7.2.1 Geometrische Interpretation

Die Punktmenge $\{0,1\}^n$ kann man sich geometrisch als n-dimensionalen Hyperwürfel veranschaulichen: Für $n = 2$ ist es ein Quadrat, für $n = 3$ ein Würfel und für $n \geq 4$ ein Hyperwürfel, der sich nur als Projektion darstellen lässt. Die Hamming-Distanz zwischen zwei Punkten entspricht dann der Anzahl der Würfelkanten auf einem kürzesten Weg zwischen den beiden Punkten. (Es gibt für $n \geq 2$ mehrere kürzeste Wege gleicher Länge.)

Jede Bewegung entlang einer Kante entspricht dem Wechsel von 0 zu 1 bzw. 1 zu 0 auf der entsprechenden Achse.

Die maximale Distanz zwischen zwei Punkten aus $\{0,1\}^n$ ist n – sie wird z. B. von den Punkten (0,0,. . . ,0) und (1,1,. . . ,1) angenommen, denn $\sum_{i=1}^{n} |1 - 0| = n$.

Abbildung 2.7 zeigt einen Würfel mit vier Punkten, die alle Hamming-Distanz 2 voneinander haben. Man kann gut erkennen, dass in jedem Fall mindestens zwei Kanten durchlaufen werden müssen, um zwei der markierten Punkte zu verbinden.

Dem entspricht, dass in den Punkten (0,0,0), (1,0,1), (1,1,0) und (0,1,1) jeweils mindestens zwei Bits gekippt werden müssen, um einen Punkt in den anderen zu überführen.

Abbildung 2.8 zeigt die Punkte (0,0,0) und (1,1,1) im gleichen Würfel mit der maximal möglichen Hamming-Distanz 3 sowie einen der kürzesten Verdindungswege der Länge 3.

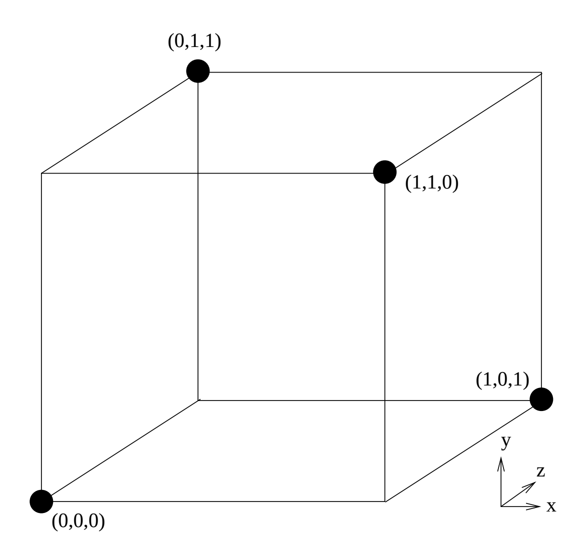


Abbildung 2.7: Hamming-Distanz 2 im Würfel.

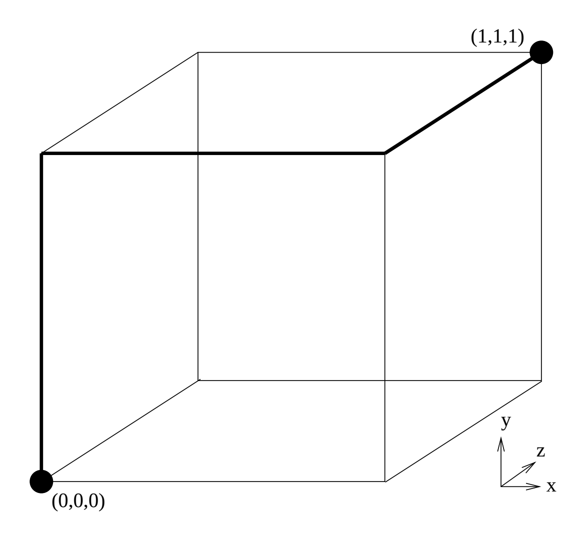


Abbildung 2.8: Hamming-Distanz 3 im Würfel – mit (einem) kürzesten Weg zwischen (0,0,0) und (1,1,1).

2.7.2.2 Hamming-Code für 6-Bit-Vektoren

Um Fehlerkorrektur eines Bitfehlers in einer 6 Bit großen Übertragung zu ermöglichen, mussten ein geeignetes n und eine Abbildung $\{0,1\}^6 \rightarrow \{0,1\}^n$ gefunden werden, so dass alle Vektoren in $Bild(f)$ (mindestens) Hamming-Distanz 3 haben.

Die Suche wurde experimentell mit Hilfe eines Python-Programms durchgeführt, das in Anhang C.5 abgedruckt ist.

Es wurde ein Code mit Hamming-Distanz 3 gefunden, der somit die Korrektur eines Übertragungsfehlers (pro 6-Bit-Block) erlaubt. Der Code ergänzt vier Paritätsbits, und alle Codes mit weniger als vier Paritätsbits haben eine Hamming-Distanz, die kleiner als 3 ist.

Die Verwendung so genannter (perfekter) Standard-Hamming-Codes war nicht möglich: Allgemein gilt für Hamming-Codes die Formel

$$d + p + 1 \leq 2^p \quad (p = \text{Anzahl der Paritätsbits}, \; d = \text{Anzahl der Datenbits})$$

Perfekte Hamming-Codes haben eine Gesamtlänge $2^n - 1$ (für sie wird die Ungleichung zur Gleichung $d + p + 1 = 2^p$), ein Beispiel dafür ist der (15,11)-Code mit elf Datenbits. Der Datenanteil hat bei solchen Codes immer die Länge $2^n - n - 1$. Damit ist keine sinnvolle Kodierung von 6-Bit-Folgen möglich, da $2^n - n - 1 \neq 6$ für alle $n \in \mathbb{N}$.

2.8 Hypertext Transfer Protocol

In RFC 2616 [FGM+99] ist die aktuelle Version des Hypertext Transfer Protocol (HTTP/1.1) beschrieben.

HTTP ist ein Anfrage-Antwort-Protokoll: Ein Client schickt eine Anfrage, die in der ersten Zeile

- die Anfragemethode (z. B. `GET`),
- einen URI (*Uniform Resource Identifer*) – das kann eine vollständige HTTP-URL oder ein absoluter Pfad auf einem an anderer Stelle angegebenen Web-Server sein – und
- die HTTP-Protokollversion (z. B. `HTTP/1.1`)

enthält; darauf folgen gegebenenfalls weitere Zeilen mit zusätzlichen Angaben; in HTTP-Version 1.1 gehört immer die Angabe des abgefragten Hosts dazu.

Der Server antwortet mit einer Statuszeile, die wiederum die Protokollversion sowie einen Erfolgs- oder Fehlercode enthält. Es folgen MIME-ähnliche Statusinformationen zum Server und zur Art der Übertragung (z. B. die verwendete Kodierung und der *Content type*) sowie schließlich die eigentlichen angeforderten Daten.

Der häufigste Fall ist der Abruf einer Web-Seite mit dem `GET`-Befehl: Dazu sendet der Client dem Server (mindestens) die zwei Zeilen

```
GET Pfad HTTP/1.1
Host: Hostname
```

Der Server antwortet darauf mit einer Nachricht, die aus zwei Hälften, Header (Kopf) und Body (Rumpf), besteht – der Header enthält, ähnlich wie E-Mail-Header, mehrere Header-Zeilen, und eine Leerzeile dient als Trennung zwischen Header und Body.

Um die volle Rückgabe des Servers (und nicht nur die als Body übertragene Web-Seite) zu sehen, kann man eines der beiden Kommandos `telnet` und `netcat` verwenden:

```
$ echo -e "GET / HTTP/1.1\nHost: hgesser.com\n" | netcat hgesser.com 80

HTTP/1.1 200 OK
Date: Sun, 08 Aug 2004 17:07:16 GMT
Server: Apache/1.3.26 (Unix) Debian GNU/Linux PHP/4.1.2
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1

ba2
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>hgesser.com: Linux-Literatur</title>
...
```

HTTP-URLs haben den allgemeinen Aufbau

```
http_URL = "http:" "//" host [ ":" port ] [ abs_path [ "?" query ]]
```

wobei `host` entweder eine IP-Adresse oder ein Rechnername ist. Die Angabe des Ports ist nicht zwingend; wird sie weggelassen, wird angenommen, dass der Port 80 gemeint ist. (HTTP-Kommunikation verwendet im Regelfall, aber nicht zwingend, TCP/IP-Verbindungen, der Standard-TCP-Port ist 80.)

Die Host-Angabe wurde in HTTP 1.1 als Erweiterung von HTTP 1.0 eingeführt, um die Möglichkeit zu schaffen, dass mehrere Domain-Namen die gleiche IP-Adresse verwenden – HTTP 1.0 ging von einer eindeutigen Zuordnung von Domain-Namen und IP-Adressen aus [FGM+99, Abschnitt 19.6.1].

`abs_path` ist eine absolute Pfadangabe in Unix-Notation, beginnt also mit einem Schrägstrich / und kann weitere als Verzeichnistrenner enthalten. Diese Anordnung der Web-Inhalte in Unterverzeichnisse muss übrigens keiner gleichartigen Verzeichnishierarchie auf dem Web-Server entsprechen.

`?query` ist eine Ergänzung der URL, die meist verwendet wird, um Parameter an CGI-Skripte zu übergeben, die dynamische Inhalte erzeugen.

Die häufigsten Erfolgs- und Fehlercodes sind 200 (OK) und 404 (*Not Found*, nicht gefunden). RFC 2616 definiert insgesamt 40 Antwortcodes.

2.8.1 TCP/IP, Fragmentierung und Segmentierung

TCP/IP bildet die Grundlage aller Internet-Dienste und ist in den RFC 791 (Internet Protocol) [Pos81b] und 793 (Transmission Control Protocol) [Pos81c] beschrieben.

IP-Pakete bestehen aus einem Header und einem Datenteil, der Header ist meist 20 Byte lang (das Header-Feld `IHL`, *IP Header Length*, gibt die exakte Länge an) und hat den in Abbildung 2.9 (auf der folgenden Seite) dargestellten Aufbau.

Das IP-Protokoll unterstützt die Fragmentierung von IP-Paketen: Beim Transport vom Sender zum Empfänger werden Pakete über mehrere Zwischenstationen geleitet. Auf den Teilstrecken sind unterschiedliche maximale Paketgrößen möglich, z. B. ist für Einwahlverbindungen eine maximale Transfergröße (MTU, *maximum*

0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Version	IHL	Type of Service	Total Length
Identification		Flags	Fragment Offset
Time to Live	Protocol	Header Checksum	
Source Address			
Destination Address			
Options			Padding

Abbildung 2.9: Aufbau des IP-Headers.

transfer unit) von 1500 Byte üblich. Die kleinste MTU auf dem Weg vom Sender zum Empfänger heißt *Path MTU* (Pfad-MTU, kurz PMTU). RFC 1191 [MD90] gibt Hinweise dazu, auf welche Weise die PMTU ermittelt werden kann, damit nur Datenpakete verschickt werden, deren Größen unterhalb der PMTU liegen.

Ist ein IP-Paket größer als für die nächste Teilstrecke zulässig, wird es in Fragmente unterteilt. Jedes Fragment ist dabei ein reguläres IP-Paket, dessen Header größtenteils mit dem Header des Originalpakets identisch ist; über die Header-Felder `More Fragments` (das ist das dritte Bit im Feld `Flags`) und `Fragment Offset` kann der Empfänger später die Fragmentierung erkennen und die Teile wieder zusammensetzen.

Ein TCP-Paket besteht ebenfalls aus Header und Daten; auch TCP-Header haben eine variable Länge und den in Abbildung 2.10 gezeigten Aufbau.

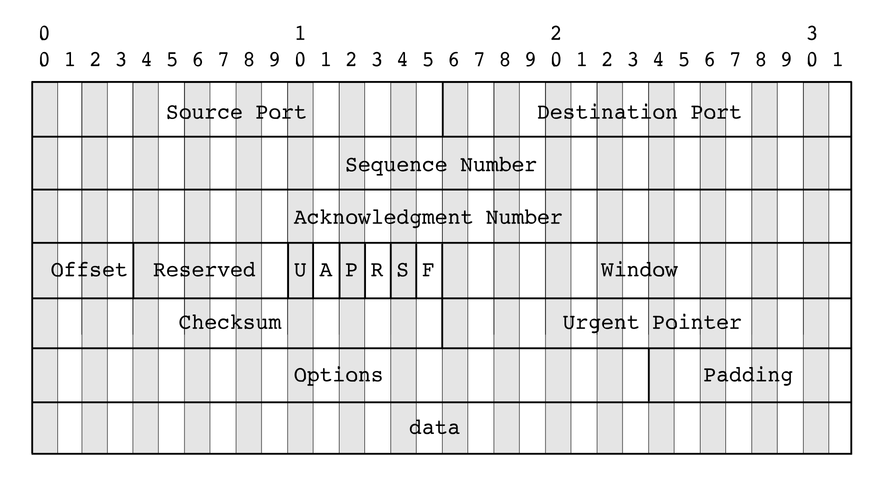


Abbildung 2.10: Aufbau des TCP-Headers.

Absender- und Empfängeradressen sind im TCP-Header nicht zu finden, da diese Daten bereits im IP-Header gespeichert werden und damit redundant wären.

TCP selbst nahm traditionell keine Fragmentierung vor, da diese Aufgabe auf IP-Ebene übernommen wird:

> The internet protocol also deals with any fragmentation or reassembly of the TCP segments required to achieve transport and delivery through multiple networks and interconnecting gateways. [Pos81c, Abschnitt 1.1]

Allerdings teilen sich die Verbindungspartner auf TCP-Ebene beim Aufbau der Verbindung (*Handshake*) mit, welche maximale Segmentgröße (*Maximum Segment Size*, kurz MSS) sie für Übertragungen akzeptieren. (Dafür wird das Header-Feld `Options` verwendet.)

Für die Berechnung der MSS gibt es keine eindeutige Richtlinie. RFC 879 [Pos83] weist darauf hin, dass der für die IP- und TCP-Header benötigte Platz zwischen 40 und 120 Byte liegen kann, dass also entsprechend die MSS zwischen $\text{MTU}-40$ und $\text{MTU}-120$ liegen kann. (Die MSS ist ein TCP-Parameter, die MTU ein IP-Parameter.)

In der Praxis wird häufig der Wert $\text{MTU}-52$ gewählt, weil viele aktuelle TCP-Implementierungen jedem Paket einen 12 Byte langen Optionenblock mit *Timestamps* mit auf den Weg geben. Das erlaubt eine bessere Berechnung der *Round-trip*-Zeiten (siehe RFC 1323, „TCP Extensions for High Performance“ [JDB92]).

Die folgenden Zeilen zeigen den `tcpdump`-Mitschnitt des *Handshake* zwischen einem Web-Server und einem Client auf dem gleichen Rechner; die MTU des Netzwerkgeräts `lo` ist 16436.

```
23:41:52.448536 localhost.51127 > localhost.http: S 2036275665:2036275665(0) win
32767 <mss 16396,sackOK,timestamp 271006675 0,nop,wscale 0> (DF)
23:41:52.448561 localhost.http > localhost.51127: S 2048690357:2048690357(0) ack
2036275666 win 32767 <mss 16396,sackOK,timestamp 271006675 271006675,nop,wscale
0> (DF)
```

Die MSS ist hier offensichtlich als 16436−40=16396 bestimmt worden.

Bei einer Internet-Verbindung kommt es zu anderen Werten – beide Rechner sind für die jeweiligen Netzwerkgeräte `eth0` mit MTU 1500 konfiguriert:

```
22:02:50.664335 server.esser.de.48258 > p15122306.pureserver.info.http: S 186453
182:186453182(0) win 5840 <mss 1460,sackOK,timestamp 270412497 0,nop,ws cale 0>
(DF)
22:02:50.737705 p15122306.pureserver.info.http > server.esser.de.48258: S 283895
4249:2838954249(0) ack 186453183 win 5792 <mss 1400,sackOK,timestamp 89 2040690
270412497,nop,wscale 0> (DF)
```

Es sieht so aus, als ob der Client die MSS durch Abzug von 40 errechnet hat und beim Server der Wert 100 abgezogen wurde – nach einem Heruntersetzen der Server-seitigen MTU auf 1200 und Wiederholung des Tests ergibt sich als MSS allerdings 1160 (also Differenz 40), so dass die MSS 1400 andere Ursachen hat.

Betrachtet man die `tcpdump`-Protokolle auf beiden Seiten, sieht man, dass Client und Server beide eine MSS von 1460 vorschlagen – beide empfangen aber die MSS 1400. „Schuld“ hat die DSL-Verbindung. Der Router setzt in beiden Richtungen die MSS-Größen in den *Handshake*-Paketen herunter, um möglichen Problemen aus dem

Weg zu gehen. Durch die von DSL verwendete Kapselung (PPPoE, *PPP over Ethernet*) gehen zusätzliche 6 Byte (für PPPoe) und weitere 2 Byte (für PPP) verloren, so dass sich die MTU von 1500 Byte auf 1492 Byte verringert. (Eine Beschreibung liefert die PPPoE-Dokumentation von Skoll [Sko00].) Die MSS nun gleich um 60 Byte zu reduzieren, ist allerdings stark konservativ. Die Vorgabe lässt sich bei manchen Routern anpassen (Abbildung 2.11), der maximal mögliche Wert ist aber 1492.

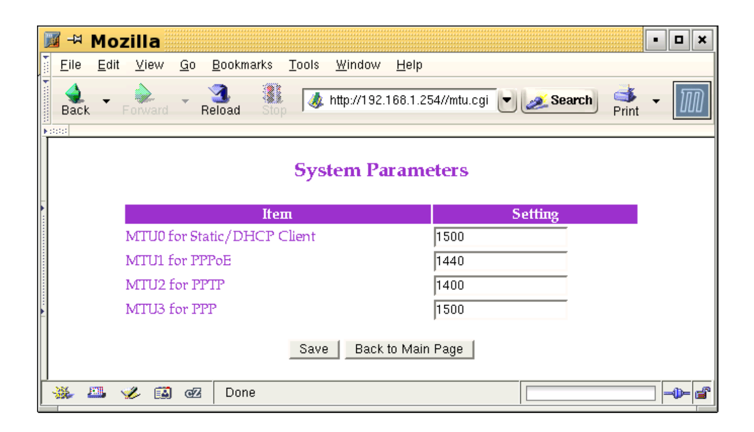


Abbildung 2.11: Einstellung der PPPoE-MTU über ein verstecktes Service-Menü.

HTTP-Pakete sind TCP-Pakete, die über IP versendet werden und damit der Fragmentierung unterliegen, wenn die Größe eines IP-Pakets auf einer Teilstrecke die dort geltende MTU überschreitet. Zur IP-Fragmentierung kommt es aber nicht, da schon auf TCP-Ebene passende Pakete erzeugt werden. Anhang E (ab Seite 135) analysiert eine HTTP-Übertragung mit Hilfe von `strace` und `ethereal`.

2.9 Statistik

Für die Auswertung von Messwerten werden einige statistische Begriffe benötigt, die im Folgenden vorgestellt werden. Dazu werden stets endliche reelle Zahlenfolgen $(t_1, t_2, \ldots, t_n) \subset \mathbb{R}$ betrachtet.

Definition

Der Durchschnitt der n Werte

$$\text{avg}((t_i)_{i=1}^n) := \frac{1}{n} \sum_{i=1}^{n} t_i$$

heißt **Mittelwert** der (t_i). Statt $\text{avg}((t_i)_{i=1}^n)$ wird auch die Kurzschreibweise $\bar{t}$ verwendet, wenn der Zusammenhang klar ist.

Der **Median** $\text{med}((t_i)_{i=1}^n)$ teilt die Messwerte in zwei gleich große Blöcke (die oberhalb bzw. unterhalb des Medians liegen). Er wird berechnet, indem die Folge sortiert wird und dann

- bei ungerader Anzahl Werte der mittlere Wert
- bei gerader Anzahl der Mittelwert der beiden mittleren Werte

bestimmt wird.

Die **Standardabweichung** σ ist die Wurzel der durchschnittlichen quadratischen Differenz der Werte zum Mittelwert:

$$\sigma((t_i)_{i=1}^n) := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(t_i - \overline{t})^2}$$

(wobei $\overline{t}$ der Mittelwert $\text{avg}((t_i)_{i=1}^n)$ ist). Ist der Zusammenhang klar, wird nur kurz σ geschrieben. σ^2 wird auch **Varianz** genannt.

2.10 Zusammenfassung

In diesem Kapitel wurden alle Grundlagen geschaffen, die benötigt werden, um die Ziele dieser Diplomarbeit zu erreichen und die Lösungswege zu beschreiben: Verdeckte Kanäle (sowohl Zeit- als auch Ressourcenkanäle) wurden definiert und an Beispielen erläutert; verwandte Techniken (Kryptographie und Steganographie) wurden mit verdeckten Kanälen verglichen, und nach möglichen Anwendungsgebieten verdeckter Kanäle wurde ein theoretisches Resultat über die Kanalkapazität spezieller verdeckter Zeitkanäle vorgestellt. Auch Grundlagen von HTTP, TCP und IP wurden behandelt.

Im folgenden Kapitel geht es nun – nach einem Trivialbeispiel für die Implementierung eines verdeckten Kanals – weiter mit einer Darstellung, wie ein Web-Server über zeitliche Verzögerungen einzelner Datenpakete mit einem Proxy-Server verdeckt kommuniziert. Die Implementierung, die aus einer Anpassung des meist verwendeten Web-Servers Apache und diversen Python-Programmen besteht, wird dort im Detail beschrieben.

Kapitel 3

Versuchsaufbau und -durchführung

Dieses Kapitel beschreibt die Implementierung eines verdeckten Kanals, der reguläre HTTP-Verbindungen zwischen einem Web-Proxy und einem Web-Server ausnutzt.

3.1 Einleitung

Nach einem einleitenden Trivialbeispiel für verdeckte Kommunikation im folgenden Kapitel 3.2 wird in Kapitel 3.3 (ab Seite 39) zunächst skizziert, wie die Datenübertragung zwischen einem modifizierten Apache-Web-Server und einem im Rahmen dieser Arbeit entwickelten Proxy-Server als Trägerkanal für verdeckte Daten ausgenutzt wird. Auf dem verdeckten Kanal wird aus Kapazitätsgründen mit einem reduzierten Alphabet (bestehend aus 64 Großbuchstaben, Ziffern und einigen Sonderzeichen) gearbeitet. Diese in 6 Bit kodierbaren Zeichen werden übermittelt, indem der Apache-Server den Transfer einer großen Datei in 4-KByte-Blöcke zerlegt, welche – abhängig vom nächsten zu sendenden Bit – verzögert (1) oder unverzögert (0) übertragen werden.

Die Verzögerungsentscheidung wird dabei in einen externen Daemon-Prozess verlagert: Ein Kontrollprogramm teilt dem Daemon die verdeckte Nachricht mit, und der Apache-Server fragt vor jedem 4-KByte-Blockversand beim Daemon nach, ob dieser nächste Block (für die zugeordnete IP-Adresse) verzögert werden soll oder nicht.

Kapitel 3.4 beschreibt ab Seite 43, welche Modifikationen am Apache-Server nötig waren und welche Zusatzprogramme (in Python) entwickelt wurden. Im darauf folgenden Unterkapitel wird das Zusammenspiel der einzelnen Komponenten untersucht.

Kapitel 3.6 (ab Seite 58) beschreibt, auf welche Weise die gemessenen Daten statistisch analysiert wurden, um zwischen verdeckt übertragenen Nullen und Einsen unterscheiden zu können.

Schließlich wird in Kapitel 3.7 ab Seite 62 ein einfaches Fehlerkorrekturverfahren vorgestellt, bei dem jeder 6-Bit-Block um vier Paritätsbits ergänzt wird, um (je Block) die Korrektur maximal eines falsch übertragenen Zeichens zu erlauben.

Mit der Auswertung der Messergebnisse für verschiedene Testszenarien beschäftigt sich Kapitel 4 ab Seite 65.

Für das bessere Verständnis wird zunächst ein Trivialbeispiel vorgestellt, bei dem von einem Server zum Client eine einfache Textbotschaft übermittelt wird.

3.2 Trivialbeispiel

Ein minimales Beispiel verdeutlicht die Kommunikation über einen verdeckten Kanal.

Auf Rechner *A* läuft ein Senderprozess *a*, der einen Verbindungsaufbau von Rechner *B* (Prozess *b*) zulässt und an ihn einen vorgegebenen Text überträgt. Verwendet werden Standard-Sockets, *a* lauscht auf Port 21000.

Die reguläre, nicht verdeckte Kommunikation kann mit einem Paket-*Sniffer* überwacht werden.

Lässt man das Skript aus Listing 3.1 laufen und baut eine Telnet-Verbindung zu Port 21000 auf, wird die Testnachricht Byte für Byte übertragen:

```
> telnet localhost 21000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Das ist eine bedeutungslose Testnachricht.
Connection closed by foreign host.
```

Um nun einen verdeckten Kanal zu integrieren, wird das Programm leicht verändert: Abhängig von einem String `secret`, der aus Nullen und Einsen besteht, wird eine künstliche Verzögerung eingeführt. Im Beispiel ist der Secret-String genauso lang wie die Testnachricht; steht an der aktuellen Position der Nachricht im Secret-String eine Eins, erfolgt vor dem Senden eine 0,1 Sekunden lange Verzögerung. Geändert wurde nur die Routine `serve()`:

```
#!/usr/bin/python
#
# trivial/sender-cc.py
...
# serve(): Routine, die mit einem Client spricht. Nachricht wird
#          Byte für Byte übertragen
def serve(c):
    secret="0000100001001000000000000100000000001000010"
    counter=0
    for x in message:
        if secret[counter] == "1":
            sleep(0.1)    # 0,1 Sekunden warten
        c.send(x)         # Zeichen senden
        counter+=1
```

Beim Telnet-Aufruf wird wie oben die Testnachricht ausgegeben; die 0,1-Sekunden-Verzögerungen sind gerade noch durch Hinsehen bemerkbar.

```
#!/usr/bin/python
#
# trivial/sender-nocc.py

# Init
from socket import *
message="Das␣ist␣eine␣bedeutungslose␣Testnachricht.\n"

# serve(): Routine, die mit einem Client spricht. Nachricht wird
#          Byte für Byte übertragen
def serve(c):
    for x in message:
        c.send(x)

# main()
s = socket(AF_INET, SOCK_STREAM)      # TCP Socket erzeugen
s.bind(("",21000))                    # An Port 21000 binden
s.listen(1)                           # Eine Verbindung zulassen

while 1:
    client,addr = s.accept()          # Verbindung annehmen
    serve(client)                     # Mit Client arbeiten
    client.close()                    # Verbindung beenden
```

Listing 3.1: Trivialbeispiel: Sender ohne verdeckten Kanal

Zur Dekodierung der verdeckten Nachricht wird ein spezielles Empfangsprogramm verwendet, das die Daten Byte-weise liest und die Verzögerungen auswertet. Daraus kann dann die verdeckte Nachricht abgeleitet werden. Listing 3.2 (auf der folgenden Seite) zeigt den Quelltext.

In der Ausgabe dieses Programms wird nach der regulären Nachricht auch die versteckte Botschaft angezeigt; zum Vergleich wird darunter die Originalnachricht ausgegeben:

```
> ./receiver-cc.py
Das ist eine bedeutungslose Testnachricht.
Secret: 0000100001001000000000000001000000000001000010
> grep "secret=" sender-cc.py | cut -c5-
secret="0000100001001000000000000001000000000001000010"
```

Geheime Botschaften können nun bitkodiert und unerkannt über diesen verdeckten Kanal übertragen werden. Die relative Kapazität des verdeckten Kanals beträgt 1 Bit je auf dem Trägerkanal gesendeten Byte.

3.3 Überblick und Entwurf der Implementierung

Um verdeckte Kommunikation in HTTP-Übertragungen einzubauen, werden auf der Senderseite ein angepasster Apache-Web-Server [Apab] und auf der Empfängerseite ein HTTP-Proxy verwendet.

Der Apache-Server auf Senderseite wird wie folgt modifiziert:

```
#!/usr/bin/python
#
# trivial/receiver-cc.py

# Init
from socket import *
from time import time
from sys import stdout
server="localhost"
port=21000

# main()
s = socket(AF_INET, SOCK_STREAM)      # TCP Socket erzeugen
s.connect((server,port))              # Verbindung zu server:21000 aufbauen

done=0
times=[]
while not done:
    t0=time()
    x = s.recv(1)
    t1=time()
    if x != '':
        stdout.write(x)               # Zeichen ausgeben
        stdout.flush()                # Nicht auf mehr Ausgaben warten
        times.append(t1-t0)           # Verzögerung speichern
    else:
        done=1
s.close()

secret=""
for t in times:
    if t > 0.01:
        secret+="1"
    else:
        secret+="0"
print "Secret:", secret
```

Listing 3.2: Trivialbeispiel: Empfänger mit verdecktem Kanal

- Während der Apache-Server eine Anfrage bearbeitet und in einer Schleife blockweise Daten überträgt, schickt er vor jedem Blockversand eine Anfrage an einen Daemon-Prozess, in der er die IP-Adresse des Rechners übermittelt, dem er das nächste Paket schicken will.
- Abhängig von der Antwort des Daemon-Prozesses (die entweder „0“ oder „1“ lautet) verzögert der Server die Übertragung des Pakets um eine festgelegte Zeit.

Es wird ein Empfänger-Proxy mit folgender Eigenschaft entwickelt:

- Beim Empfang wird nach verdeckten Botschaften (durch Zeitverzögerungen) gesucht, indem die Zeiten zwischen der Anfrage eines weiteren Datenblocks und dessen Auslieferung gemessen werden. Der Proxy schreibt die Messwerte für die spätere Analyse in eine Protokolldatei.

Um nur eine möglichst geringe Menge an Bits übertragen zu müssen, wurde der Bereich der zulässigen Zeichen eingeschränkt: Mit der implementierten Software können nur ASCII-Zeichen zwischen 32 und 95 übertragen werden, das sind die Großbuchstaben A–Z, die Ziffern 0–9 und einige Sonderzeichen (siehe Tabelle 3.1).

32 :	48 : 0	64 : @	80 : P
33 : !	49 : 1	65 : A	81 : Q
34 : "	50 : 2	66 : B	82 : R
35 : #	51 : 3	67 : C	83 : S
36 : $	52 : 4	68 : D	84 : T
37 : %	53 : 5	69 : E	85 : U
38 : &	54 : 6	70 : F	86 : V
39 : '	55 : 7	71 : G	87 : W
40 : (	56 : 8	72 : H	88 : X
41 :)	57 : 9	73 : I	89 : Y
42 : *	58 : :	74 : J	90 : Z
43 : +	59 : ;	75 : K	91 : [
44 : ,	60 : <	76 : L	92 : \
45 : -	61 : =	77 : M	93 :]
46 : .	62 : >	78 : N	94 : ^
47 : /	63 : ?	79 : O	95 : _

Tabelle 3.1: Die ASCII-Tabelle von 32 bis 95.

Kleinbuchstaben werden automatisch in Großbuchstaben umgewandelt, sonstige Zeichen werden zu Leerzeichen. Durch die Beschränkung auf 64 mögliche Zeichen ist ein Zeichen in 6 Bit kodierbar, gegenüber einem vollen Byte wird also eine Einsparung von $1-6/8 = 25\%$ erreicht. Jede Nachricht erhält – zur besseren Erkennung – Anfangs- und Endmarkierungen: Der Nachrichtenanfang wird durch den *Header* „_ “ (ASCII 95, 32: Unterstrich und Leerzeichen), das Nachrichtenende durch den *Footer* „ _“ (ASCII 32, 95) gekennzeichnet.

3.3.1 Steuerung des Servers

Der modifizierte Apache-Server wird über ein Steuerungsprogramm kontrolliert, die Kommunikation läuft dabei über einen zusätzlichen Daemon-Prozess, der im Hintergrund sowohl Anfragen des Apache-Server als auch des Steuerungsprogramms beantwortet.

Erster Ansatz: Named Pipes

Im ursprünglichen Entwurf war die Verwendung von *Named Pipes* vorgesehen: *Named Pipes* sind FIFOs (*First in, first out*), die im Dateisystem eines Unix-Rechners mit `mkfifo` angelegt werden. Sie können wie Dateien beschrieben und gelesen werden.

Für jeden Client, mit dem der Apache-Server kommunizieren soll, sollten in einem festgelegten Verzeichnis zwei Named Pipes `IP-Adresse.in` und `IP-Adresse.out` erzeugt werden, über die ein bidirektionaler Kanal zwischen Apache-Server und Steuerungsprogramm aufgebaut worden wäre:

- Über die *Named Pipe* `IP-Adresse.in` erhielte der Apache-Server die an den Client zu übertragenden Daten, die bereits bitkodiert wären.
- Über die *Named Pipe* `IP-Adresse.out` gäbe der Apache-Server Statusinformationen zurück, z. B., dass die Datenübertragung abgeschlossen wurde, der Client nicht erreichbar ist etc.

Named Pipes lassen sich aber nur synchron nutzen: Lese- und Schreibprozess müssen gleichzeitig auf die *Pipe* zugreifen, ansonsten blockiert der „partnerlose“ Prozess.

Die Alternative wäre die Simulation einer *Pipe* über eine Datei: In diese könnte das Kontrollprogramm schreiben, und der Apache-Server würde sie erst dann auslesen, wenn er sie benötigt. Der Leseprozess müsste aber die Dateien von vorne abschneiden – diese Dateioperation lässt sich nicht effizient implementieren. (Wenn man eine Maximalgröße für den Puffer definiert, ist eine effiziente Implementierung über Dateien möglich: Die ersten vier Bytes der Datei enthalten die aktuellen Lese- und Schreibpositionen, der eigentliche Inhalt folgt ab dem fünften Byte. Wird beim Lesen oder Schreiben die letzte zulässige Position in der Datei erreicht, wird am Anfang weiter gelesen bzw. geschrieben.)

Die Nutzung von *Pipes* für die Datenübertragung hätte den Vorteil geboten, dass kein lokaler Netzwerkverkehr stattfände, der mit einem lokal eingesetzten Netzwerk-*Sniffer* beobachtet werden könnte; allerdings kann davon ausgegangen werden, dass der Server ohnehin in der Hand des Angreifers ist – auch der Start zusätzlicher Prozesse hinterlässt Spuren: So tauchen zum Beispiel alle Programme in der mit `ps` erhältlichen Prozessliste auf, sofern diese nicht mit einem Hilfsprogramm versteckt werden, wie es etwa bei vielen *Root kits* Standard ist.

Zweiter Ansatz: Daemon-Modell

Im Daemon-Modell funktioniert die Kommunikation wie folgt:

1. Wenn der Apache-Server eine HTTP-Anfrage von der IP-Adresse n erhält, schickt er an den Daemon die Nachricht „A n“.
2. Der Daemon interpretiert die Nachricht als Anfrage, ob der Client mit IP-Adresse n an der verdeckten Kommunikation teilnimmt. Wird diese Frage verneint, sendet der Daemon die Antwort 0. Wird sie bejaht, schickt er als Antwort entweder 0 oder 1.
3. Der Apache-Server verzögert die Kommunikation, wenn er eine 1 empfängt, anderenfalls nicht.

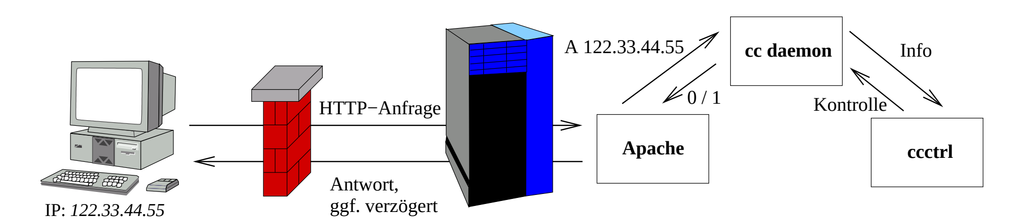


Abbildung 3.1: Kommunikation zwischen Client und Server sowie lokal auf dem Server.

Außerdem beantwortet der Daemon Statusabfragen des Kontrollprogramms und nimmt von diesem auch Befehle zur Aufnahme einer neuen Client-IP-Adresse sowie die zu sendenden Daten entgegen. Abbildung 3.1 zeigt die Kommunikation zwischen allen Komponenten.

Der Daemon implementiert ein eigenes *Pipe*-Modell im Hauptspeicher: Für jede IP-Adresse eines Rechners, mit dem verdeckt kommuniziert werden soll, speichert er eine aus Nullen und Einsen bestehende Zeichenkette, die von hinten verlängert und von vorne verkürzt werden kann.

3.4 Details der Implementierung

Für die Empfängerseite wurde ein einfacher HTTP-Proxy in Python geschrieben, auf der Senderseite wurde der HTTP-Server Apache [Apab] in Version 1.3.31 modifiziert. Dabei musste beachtet werden, dass die via Netzwerküberwachung beobachtbare Kommunikation inhaltlich nicht verändert wird.

Mit Ausnahme der Modifikationen des Apache-Servers wurden alle weiteren Programme in der Skriptsprache Python geschrieben: Python ist sehr mächtig, verwendet eine gut lesbare, intuitive Syntax und eignet sich für die schnelle Programmentwicklung.

Die Programmquelltexte und Patches für Apache sind auf der Web-Seite des Autors [Eße05] und auf der dieser Arbeit beiliegenden CD verfügbar.

Eine umfassende Bedienungsanleitung, anhand der die Messungen nachvollzogen werden können, bietet Anhang B ab Seite 105.

3.4.1 Client-Proxy

Als Teil dieser Arbeit wurde ein simpler HTTP-Proxy in Python implementiert. Er ist in seiner Funktion sehr beschränkt, zum Beispiel verarbeitet er immer nur eine Anfrage gleichzeitig.

Der Proxy liest nach Anforderung der Web-Seite blockweise die übertragenen Daten ein und speichert, wie viel Zeit zwischen Anforderung und Auslieferung eines Blocks liegt.

Der vollständige Quell-Code des Client-Proxys ist in Anhang C.2 (ab Seite 120) abgedruckt; von besonderem Interesse ist die Berechnung der Verzögerungen:

```
# Blockweise holen und zustellen
done=0
psize=4096                          # Blockgröße für recv(): 4 KByte
times=[]                            # Hier die Wartezeiten speichern
addedtime = 0.0                     # Evtl. mehrere recv()-Aufrufe
recvsize = 0                        # Auf 4096 hochzählen
while not done:
    t0=time()
    result=s_web.recv(psize-recvsize)
    t1=time()
    recvsize += len(result)
    addedtime += (t1-t0)
    if recvsize >= psize:
        times.append(addedtime)     # Verzögerung speichern
        recvsize -= psize
        addedtime = 0.0
    c.send(result)
    if result=="":
        done=1
s_web.close()
```

In `addedtime` werden die (als Differenz von `t1` und `t0`) gemessenen Übertragungszeiten aufaddiert, bis ein vollständiger 4-KByte-Block gelesen wurde. Diese aufsummierten Zeiten werden im Array `times` gespeichert und am Ende der Übertragung in eine Protokolldatei geschrieben.

Bemerkung

Der hier implementierte Proxy ist im Sinne des HTTP-Protokolls, RFC 2616 [FGM+99], ein *transparenter Proxy*, das heißt, er erfüllt die Anforderung, die Daten nicht zu verändern:

> A ‘transparent’ proxy is a proxy that does not modify the request or response beyond what is required for proxy authentication and identification.

3.4.2 Modifikation des Apache-Servers

Der komplexere Teil der Implementierung ist die Anpassung des Apache-Web-Servers. Der Quellcode der Apache-Version 1.3.31 ist im Archivbereich der Apache-Homepage [Apaa] erhältlich und liegt auch auf der dieser Arbeit beiliegenden CD.

Am Anfang der Implementierungsarbeit stand eine Untersuchung der Apache-Quellen daraufhin, welche Funktion für das Verschicken der Datenpakete zuständig ist. Aus der Apache-API-Dokumentation [Apac] ergaben sich einige Kandidaten, darunter u. a. `send_fd_length` und `send_fb_length`. Mit Hilfe einer Protokollfunktion wurden einige Funktionen im laufenden Betrieb überwacht.

3.4.2.1 Logger für Apache-Funktionsaufrufe

Die Funktion `ap_covert_log` wurde in `http_protocol.c` ergänzt. Über Aufrufe an verschiedenen Stellen des Apache-Quellcodes wurde nach der Funktion gesucht, die die Pakete an den Client schickt.

```
/* ap_covert_log schreibt in eine Log-Datei */

void ap_covert_log (char *message) {
  int fd = fopen ("/tmp/ap_covert.log","a");   /* Log-Datei öffnen */
  fprintf (fd, message);                       /* Log-String schreiben */
  fclose (fd);
}
```

3.4.2.2 Die Apache-Funktionen zum Senden

Apache verwendet in Version 1.3.31 verschiedene Funktionen zum Versenden von Datenpaketen. In der Regel überträgt es diese mit `ap_send_fd` (in der Datei `http_protocol.c`), und diese Funktion ruft `ap_send_fd_length` auf. Der relevante Ausschnitt der Funktion ist in Listing 3.3 (auf der folgenden Seite) zu sehen.

Der Aufruf von `ap_bwrite()` in Zeile 31 ist die Stelle, an der die Verzögerungen eingebaut werden.

Dateien, die durch das Apache-Modul `mod_gzip` geschleust werden, überträgt der Apache mit `ap_send_mmap` (auch in der Datei `http_protocol.c`), die den gleichen groben Aufbau mit zwei Schleifen hat (Listing 3.4 auf der übernächsten Seite). Auch hier ist wieder vor dem Aufruf von `ap_bwrite()` (Zeile 21) der Code für die Verzögerungen einzubauen.

Um die IP-Adresse zu finden, untersucht man den Datentyp `conn_rec`, der in der Datei `src/include/httpd.h` beschrieben ist:

```
struct conn_rec {
    [ ... ]
    /* Who is the client? */

    struct sockaddr_in local_addr;      /* local address */
    struct sockaddr_in remote_addr;     /* remote address */
    char *remote_ip;            /* Client's IP address */
    char *remote_host;          /* Client's DNS name, if known.
    [ ... ]
};
```

Die IP-Adresse wird demnach in `char *remote_ip` gespeichert, so dass man sie innerhalb der Funktion `ap_send_mmap` über `r->connection->remote_ip` abrufen kann.

```
1  /*
2   * Send the body of a response to the client.
3   */
4  API_EXPORT(long) ap_send_fd(FILE *f, request_rec *r)
5  {
6      return ap_send_fd_length(f, r, -1);
7  }
8
9  API_EXPORT(long) ap_send_fd_length(FILE *f, request_rec *r, long length)
10 {
11     char buf[IOBUFSIZE];
12
13     [ ... ]
14
15     while (!r->connection->aborted) {
16         if ((length > 0) && (total_bytes_sent + IOBUFSIZE) > length)
17             len = length - total_bytes_sent;
18         else
19             len = IOBUFSIZE;
20
21         while ((n = fread(buf, sizeof(char), len, f)) < 1
22                && ferror(f) && errno == EINTR && !r->connection->aborted)
23             continue;
24
25         if (n < 1) {
26             break;
27         }
28         o = 0;
29
30         while (n && !r->connection->aborted) {
31             w = ap_bwrite(r->connection->client, &buf[o], n);
32
33             [ ... ]
34         }
35     }
36
37     ap_kill_timeout(r);
38     SET_BYTES_SENT(r);
39     return total_bytes_sent;
40 }
```

Listing 3.3: Ausschnitt aus der Funktion ap_send_fd()

```
 1 #ifndef MMAP_SEGMENT_SIZE
 2 #define MMAP_SEGMENT_SIZE       32768
 3 #endif
 4 
 5 /* send data from an in-memory buffer */
 6 API_EXPORT(size_t) ap_send_mmap(void *mm, request_rec *r, size_t offset,
 7                                 size_t length)
 8 {
 9     [ ... ]
10 
11     length += offset;
12     while (!r->connection->aborted && offset < length) {
13         if (length - offset > MMAP_SEGMENT_SIZE) {
14             n = MMAP_SEGMENT_SIZE;
15         }
16         else {
17             n = length - offset;
18         }
19 
20         while (n && !r->connection->aborted) {
21             w = ap_bwrite(r->connection->client, (char *) mm + offset, n);
22 
23             [ ... ]
24         }
25     }
26 
27     ap_kill_timeout(r);
28     SET_BYTES_SENT(r);
29     return total_bytes_sent;
30 }
```

Listing 3.4: Ausschnitt aus der Funktion `ap_send_mmap()`

3.4.2.3 Eine neue sleep-Funktion

Aus dem Buch von Mitchell, Samuel und Oldham [MSO01] wurde die Funktion better_sleep übernommen, die als Argument eine Realzahl erlaubt. (sleep(3) lässt nur ganzzahlige Argumente zu.)

```
int better_sleep (double sleep_time) {
  /* QUELLENANGABE: Übernommen aus
     http://www.quepublishing.com/articles/article.asp?p=23618&seqNum=11
     "Advanced Linux Programming", Mark Mitchell & Jeffrey Oldham [MSO01] */
  struct timespec tv;
  /* Construct the timespec from the number of whole seconds... */
  tv.tv_sec = (time_t) sleep_time;
  /* ... and the remainder in nanoseconds. */
  tv.tv_nsec = (long) ((sleep_time - tv.tv_sec) * 1e+9);
  while (1) {
    /* Sleep for the time specified in tv. If interrupted by a
       signal, place the remaining time left to sleep back into tv. */
    int rval = nanosleep (&tv, &tv);
    if (rval == 0)
      /* Completed the entire sleep time; all done. */
      return 0;
    else if (errno == EINTR)
      /* Interrupted by a signal. Try again. */
      continue;
    else
      /* Some other error; bail out. */
      return rval;
  }
  return 0;
}
```

better_sleep nutzt die Systemfunktion nanosleep(2), die mit Sekunden und Nanosekunden (Milliardstel Sekunden: 10^9 ns = 1 s) arbeitet. Alternativ könnte auch die Funktion usleep(3) verwendet werden, die ihr Argument als Angabe der Mikrosekunden interpretiert und entsprechend lange wartet.

3.4.2.4 Implementierung der Verzögerungen

MMAP_SEGMENT_SIZE erhält in http_protocol.h den Wert 32768, die Größe eines (von ap_send_mmap) übertragenen Blocks ist also 32 KByte. Entsprechend legt die Präprozessor-Variable IOBUFSIZE die Größe der von ap_send_fd übertragenen Blöcke fest. Sie wird in httpd.h definiert, der Vorgabewert in den unveränderten Apache-Quellen ist 8192 Byte (8 KByte).

Tests haben gezeigt, dass diese Blockgröße bis auf 4 KByte reduziert werden kann; bei kleineren Größen werden dennoch 4-KByte-Blöcke übertragen.

Die entscheidende Anpassung der Funktionen ap_send_fd und ap_send_mmap erfolgt in der inneren while-Schleife ab while (n && !r->connection->aborted) – die geänderte Fassung sieht (im Fall von ap_send_fd) wie folgt aus:

```
while (n && !r->connection->aborted) {

    /* begin covert */
    covert_ip = r->connection->remote_ip;
    if (ap_covert_check_ip (covert_ip)) {
        better_sleep(AP_CC_SLEEPTIME);
        /* Schlafenszeit AP_CC_SLEEPTIME in
           include/covert_channel.h definiert */
    }
    /* end covert */

    w = ap_bwrite(r->connection->client, (char *) mm + offset, n);
    [...]
}
```

Die Anpassung von `ap_send_mmap` geschieht in gleicher Weise an der entsprechenden Stelle.

Vor jedem Versand eines Pakets wird die Funktion `ap_covert_check_ip()` aufgerufen, die beim Daemon nachfragt, ob für diese IP-Adresse die Übertragung verzögert werden soll. (Der Quellcode dieser Funktion basiert auf einer modifizierten Variante des Beispielprogrammes `tcpClient.c` von Pont [CSo04].) Bei einer positiven Antwort wird `better_sleep()` aufgerufen.

Alle veränderlichen Parameter sind in der Datei `src/include/covert_channel.h` definiert:

```
/*
 * covert_channel.h: Definitions for covert channels
 */

/* wie lange schlafen? */
#define AP_CC_SLEEPTIME 0.1

/* Daemon: IP-Adresse und Port */
#define AP_CC_HOSTNAME "192.168.1.1"
#define AP_CC_PORT 4999
```

Die *Precompiler*-Direktiven haben folgende Bedeutungen:

- `AP_CC_PORT` ist der Port, auf dem der Daemon Anfragen entgegen nimmt.
- `AP_CC_HOSTNAME` enthält die IP-Adresse des Rechners, auf dem der Daemon läuft.
- `AP_CC_SLEEPTIME` legt fest, wie lang die Verzögerungen sind. Die Vorgabe von 0,1 s aus den ersten Tests kann im lokalen Fall und im lokalen Netzwerk deutlich reduziert werden – wie weit, hängt von den Bedingungen im Netzwerk ab und ist Gegenstand der Untersuchungen in Kapitel 4 ab Seite 65.

`ap_covert_check_ip()` erhält als Argument die IP-Adresse des Clients, mit dem der Apache-Server gerade in Verbindung steht – er setzt daraus die Anfrage

```
A IP-Adresse
```

zusammen, schickt Sie an den Daemon, wertet die Antwort (0 oder 1) aus und gibt entsprechend den Integer-Wert 0 oder 1 zurück:

```
int ap_covert_check_ip (char *ip) {
  [...]
  /* "A " + IP-Adresse + Newline an Daemon senden */
  strcpy (daemon_request, "A␣");
  strcat (daemon_request, ip);
  strcat (daemon_request, "\n");
  send (sd, daemon_request, strlen(daemon_request)+1, 0);

  /* Antwort empfangen und auswerten, Verbindung schließen */
  const int MAX_LEN=2;
  recv (sd, daemon_reply, MAX_LEN, 0);
  close (sd);
  int reply;
  if (daemon_reply[0] == '1')
    { reply = 1; }   /* nur 1, wenn "1\n" empfangen wurde */
  else
    { reply = 0; }

  return reply;
}
```

Der vollständige Quelltext der Funktion `ap_covert_check_ip()` ist in Anhang C.4 auf Seite 128 zu finden.

3.4.3 Design des Daemons

Die Kommunikation zwischen dem modifizierten Apache-Server und dem Kontrollprogramm läuft über einen Daemon-Prozess ab, der auf Anfragen dieser beiden Komponenten wartet:

- Apache fragt vor jedem Versand eines Pakets beim Daemon nach, ob dieses Paket verzögert werden soll – dabei übermittelt der Apache nur die IP-Adresse, die Verwaltungsaufgaben der verdeckten Kommunikation übernimmt also der Daemon.
- Das Kontrollprogramm informiert den Daemon darüber, welche Botschaften an welche IP-Adressen geschickt werden sollen. Ferner kann der Daemon dem Kontrollprogramm Statusinformationen übermitteln.

Die Kommunikation zwischen allen Komponenten erfolgt über festgelegte Protokolle, so dass die Einzelkomponenten ausgetauscht werden können, beispielsweise gegen optimierte Varianten.

Für die Beispiel-Implementierung wurde auf eine Optimierung verzichtet, insbesondere ist das Speicher-Management des Daemons verbesserungsfähig. Der Daemon wurde in Python programmiert.

Der Daemon verwaltet ein Python-*Dictionary* von Kommunikationspartnern – ein *Dictionary* ist ein Python-Datentyp, der in einer ungeordneten Liste Schlüssel-Wert-Paare { *key* : *value* } speichert und damit eine *Hash*-Funktion erlaubt: Einträge des *Dictionarys* lassen sich über Schlüssel schnell hinzufügen und abfragen; es gibt auch eine Methode `has_key()`, die prüft, ob zu einem Schlüssel bereits ein Eintrag

vorhanden ist. Der Schlüssel ist in der Daemon-Implementierung die IP-Adresse eines Kommunikationspartners.

Jeder Eintrag des *Dictionarys* speichert Statusinformationen und die noch zu übermittelnden Nachrichten.

```
comlist = dictionary { ip_address -> com(ip_adress) }
com (ip_address) = [ status, message ]
status = [ bytes_sent, total_msg_length, start_transmission, last_access,
           bit_count ]
message = "message_string"
```

`bit_count` zählt die übertragenen Bits; jedes Zeichen der Originalnachricht wird in 6 Bit kodiert, und während die Übertragung läuft, sind einzelne ursprüngliche Zeichen unvollständig übertragen, wenn bereits die ersten aber nicht alle 6 Bit versandt wurden. Über den Bitzähler kann man bei Statusabfragen unvollständige Zeichen am Anfang der Nachricht aussortieren.

Der vollständige und kommentierte Quelltext des Daemons `ccdaemon` ist in Anhang C.1 ab Seite 111 zu finden.

Eine mögliche Erweiterung des Daemons ist, ihn auch für die Verwaltung der Verzögerungszeit zu verwenden. In der aktuellen Implementation muss der Apache-Server für jede Änderung der Verzögerungszeit neu übersetzt werden. Die verbesserte Version des Daemons würde statt einer „1“ beispielsweise die gewünschte Verzögerungszeit in Mikrosekunden an den Apache-Server senden.

3.4.3.1 Operationen des Daemons

Der Daemon kennt die folgenden Operationen auf den internen Daten:

- **Kommunikationspartner hinzufügen**

  Parameter: IP-Adresse

  Effekt: Fügt dem *Dictionary* einen neuen Kommunikationspartner hinzu, falls die IP-Adresse noch nicht vorhanden ist.

  Rückgabe: `true`, wenn die IP-Adresse noch nicht im *Dictionary* enthalten war, ansonsten `false`

- **Kommunikationspartner entfernen**

  Parameter: IP-Adresse

  Effekt: Löscht den Kommunikationspartner aus dem *Dictionary*, falls die IP-Adresse vorhanden ist.

  Rückgabe: [Status, Restnachricht], falls die IP-Adresse im *Dictionary* enthalten ist, ansonsten `false`

- **Statusabfrage**

  Parameter: IP-Adresse

Effekt: –

Rückgabe: [Status, Restnachricht], falls die IP-Adresse im *Dictionary* enthalten ist, ansonsten `false`

- **Nachricht hinzufügen**

Parameter: IP-Adresse, Nachricht

Effekt: Wenn die IP-Adresse im *Dictionary* enthalten ist, schreibt der Daemon die Nachricht in die Warteschlange.

Rückgabe: `true`, wenn die IP-Adresse im *Dictionary* enthalten ist; ansonsten `false`

Beim Hinzufügen wird die Nachricht in 6-Bit-Werte gewandelt; das erledigt die folgende Python-Funktion:

```
def cc_ascii_to_bits (s):
    result = "111111000000"         # Start-Sequenz: 111111000000
    s = upper(s)
    for c in s:
        if c < chr(32) or c > chr (95): c = chr(32)
        v = ord(c)-32               # i jetzt zwischen 0 und 63
        for i in [5,4,3,2,1,0]:
            if v & (2**i):
                result=result+"1"
            else:
                result=result+"0"
    result = result+"000000111111"  # End-Sequenz: 000000111111
    return result
```

Auch die umgekehrte Richtung kann leicht berechnet werden. Hier wird zusätzlich angegeben, wie viele Bits eines (6-Bit-) Zeichens am Anfang des Bit-Strings ignoriert werden sollen:

```
def cc_bits_to_ascii (s,index):
    result = ""
    pos = 0
    v = 0
    if index != 0: result=result+"?"
    for c in s[index:]:
        if c == "1": v = v+2**(5-pos)
        pos = pos+1
        if pos == 6:
            pos = 0
            result = result + chr (v+32)   # Zeichen anhängen
            v = 0
    return result
```

- **Bit verdeckt senden**

Parameter: IP-Adresse

Effekt: Wenn die IP-Adresse im *Dictionary* enthalten ist, entfernt der Daemon das vorderste Bit der Nachricht und aktualisiert die Statusinformationen.

Rückgabe: `"1"`, falls die IP-Adresse im *Dictionary* enthalten ist, die zugehörige Nachricht nicht leer ist und deren erstes Bit eine 1 ist; anderenfalls `"0"`

3.4.4 Kommunikation mit dem Daemon

Um den Daemon zu testen, schickt man ihm via `telnet` oder mit `netcat` Kommandos. Diese haben den Aufbau

- `C` *cmd* [*ip*] [*args*] – für Kontrollprogramm-Kommunikation,
- `A` *ip* – für Apache-Kommunikation.

Im Detail versteht der Daemon folgende Anweisungen:

- `C add` *ip*: IP-Adresse hinzufügen
- `C msg` *ip Nachricht*: Nachricht in die FIFO-Liste für die angegebene IP-Adresse eintragen
- `C msgfile` *ip Dateiname*: Nachricht aus Datei lesen und in die FIFO-Liste für die angegebene IP-Adresse eintragen
- `C stat` *ip*: Status zu IP-Adresse ausgeben
- `C allstat`: Alle Statusinformationen ausgeben
- `C del` *ip*: IP-Adresse entfernen
- `A` *ip*: Nächstes Bit der Nachricht für diese IP-Adresse abrufen

Das Kontrollprogramm schickt dem Daemon Befehle zum Eintragen einer neuen IP-Adresse, Löschen einer Adresse, Eintragen einer Nachricht in die Warteschlange und zur Statusabfrage:

```
> echo "C add 192.168.1.19" | netcat localhost 4999
OK
> echo "C stat 192.168.1.19" | netcat localhost 4999
Info for IP address 192.168.1.19
Chars sent:  0,0
Msg length:  0
Start Trans: --
Last access: --
Message:
> echo "C stat 192.168.1.20" | netcat localhost 4999
Error: IP address 192.168.1.20 not found
> echo "C msg 192.168.1.19 Testnachricht" | netcat localhost 4999
OK
> echo "C stat 192.168.1.19" | netcat localhost 4999
Info for IP address 192.168.1.19
Chars sent:  0,0
Msg length:  13
Start Trans: Sun Oct 24 14:44:42 2004
Last access: --
Message:     Testnachricht
> echo "C add 192.168.1.20" | netcat localhost 4999
OK
> echo "C allstat" | netcat localhost 4999
Info for IP address 192.168.1.19
Chars sent:  0,0
Msg length:  13
Start Trans: Sun Oct 24 14:44:42 2004
```

```
Last access: --
Message:     Testnachricht
Info for IP address 192.168.1.20
Chars sent:  0,0
Msg length:  0
Start Trans: --
Last access: --
Message:
> echo "C del 192.168.1.19" | netcat localhost 4999
Info for IP address 192.168.1.19
Chars sent:  0,0
Msg length:  13
Start Trans: Sun Oct 24 14:44:42 2004
Last access: --
Message:     Testnachricht
IP removed
```

Der Apache-Server schickt nur Anfragen mit einer IP-Adresse und erwartet als Antwort eine Null oder eine Eins:

```
> echo "A 192.168.1.19" | netcat localhost 4999
0
```

3.4.5 Design des Kontrollprogramms

Als Kontroll-Software für den modifizierten Apache-Server dient das Kommandozeilen-Tool `ccctrl` (*covert channel control*).

Das Programm versteht die folgende Syntax:

- Anlegen eines neuen Kommunikationspartners über die IP-Adresse:
  `ccctrl add` *`IP-Adresse`* (*add*)
  Es wird eine neue Warteschlange erzeugt. Das Programm gibt nur „OK“ aus – im Fehlerfall erscheint ein Hinweis, dass das Anlegen nicht möglich war.

- Löschen eines Kommunikationspartners über die IP-Adresse:
  `ccctrl del` *`IP-Adresse`* (*delete*)
  Beim Löschen wird die Warteschlange ausgelesen, das Programm gibt als Statusmeldung aus, welcher Teil der Nachricht noch nicht übertragen wurde. Danach wird die Warteschlange entfernt.

- Gesamtstatus:
  `ccctrl allstat`
  Das Programm listet alle Kommunikationspartner und statistische Angaben zu den Übertragungen auf.

- Einzelstatus eines Kommunikationspartners über die IP-Adresse:
  `ccctrl stat` *`IP-Adresse`*
  Das Programm gibt statistische Informationen zur Kommunikation mit der angegebenen IP-Adresse aus.

- Nachricht senden:
  `ccctrl msg` *`IP-Adresse Nachricht`*
  Das Programm übermittelt dem Daemon die Nachricht *Nachricht* für die angegebene IP-Adresse.

- Datei senden:
  `ccctrl msgfile` *IP-Adresse Dateiname*
  Das Programm übermittelt dem Daemon die in der Datei *Dateiname* abgelegte Nachricht für die angegebene IP-Adresse.

Die Syntax für `ccctrl` wurde so gewählt, dass das Programm besonders leicht implementiert werden kann – tatsächlich besteht `ccctrl` nur aus folgenden Zeilen:

```
#!/bin/sh
CCDAEMON=192.168.1.1
CCPORT=4999
ARGS=$@
echo "C␣"${ARGS} | netcat $CCDAEMON $CCPORT
```

Noch kürzer ist die Darstellung als Shell-Funktion, wenn man auf die Verwendung von Variablen verzichtet:

```
ccctrl () {
    echo "C␣"$@ | netcat 192.168.1.1 4999
}
```

3.5 Zusammenarbeit der Komponenten

Dieses Unterkapitel stellt die Anwendung und Zusammenarbeit der einzelnen Komponenten dar. Zunächst wird die im Daemon `ccdaemon` implementierte FIFO beschrieben, anschließend die Zusammenarbeit von Daemon und modifiziertem Apache-Server sowie die Verständigung zwischen Daemon und Kontrollprogramm.

In den ersten Tests wurde ein recht hoher Wert für die Verzögerungszeit gewählt, so dass eine klare, fehlerfreie Übertragung zu erwarten war – zeitliche Optimierungen behandelt das folgende Kapitel 4 ab Seite 65.

3.5.1 FIFO-Funktionalität des Daemons

Mit dem `add`-Kommando wird eine neue IP-Adresse für die verdeckte Kommunikation in die Liste der Kommunikationspartner eingetragen. Dabei wird eine leere FIFO initialisiert. Dies wird mit dem `stat`-Kommando überprüft:

```
> echo "C add 127.0.0.1" | netcat server 4999
OK
> echo "C stat 127.0.0.1" | netcat server 4999
Info for IP address 127.0.0.1
Chars sent:  0,0
Msg length:  0
Start Trans: --
Last access: --
Message:
```

Die IP-Adresse ist dem Daemon nun bekannt.

Im nächsten Schritt wird über den `msg`-Befehl eine Nachricht für diese IP-Adresse hinzugefügt. Diese wird vom Daemon in Großbuchstaben (zwischen ASCII 32 und 95) umgewandelt und gespeichert:

```
> echo "C msg 127.0.0.1 Mit dieser Testnachricht wird das System getestet" | \
  netcat server 4999
OK
> echo "C stat 127.0.0.1" | netcat server 4999
Info for IP address 127.0.0.1
Chars sent:  0,0
Msg length:  57
Start Trans: Sun Oct 31 12:59:48 2004
Last access: --
Message:     _ MIT DIESER TESTNACHRICHT WIRD DAS SYSTEM GETESTET _
```

3.5.2 Kommunikation zwischen Daemon, Apache und Kontrollprogramm

Der Apache-Server und der Daemon-Prozess laufen auf dem gleichen Rechner – deshalb kommt es nicht zu nennenswerten Verzögerungen: Ein ungepatchter Apache-Server hat die gleichen Reaktionszeiten und Übertragungsraten wie der gepatchte Apache-Server (wenn keine verdeckten Nachrichten übertragen werden).

Die Kommunikation zwischen Daemon und Kontrollprogramm läuft genauso ab wie die direkte Ansteuerung des Daemons über `netcat` oder `telnet`.

3.5.3 Zusammenspiel aller Komponenten

Schließlich wird der gemeinsame Einsatz aller Bestandteile betrachtet: Der HTTP-Client `wget` schickt über den Proxy `ccproxy` eine Anfrage an den Apache-Server, welcher diese auf Anweisungen vom Dameon `ccdaemon` hin teilweise verzögert.

Dazu werden zunächst auf der Server-Seite der Apache und der Daemon gestartet. Die Client-IP-Adresse wird in die Liste der Kommunikationspartner eingetragen, und eine Testnachricht wird in die FIFO-Liste geschrieben. Der Apache-Server hält eine größere Datei zum Download bereit, die dann mit `wget` heruntergeladen wird.

Damit `wget` einen Proxy-Server verwendet, muss vor dem Aufruf die Umgebungsvariable `http_proxy` (in Kleinbuchstaben) passend gesetzt werden, der Standard-Port des Proxys ist 21000:

```
export http_proxy="http://localhost:21000"
```

Zusätzlich muss beim `wget`-Aufruf die Option `--proxy=on` angegeben werden.

Der Proxy schreibt die gemessenen Verzögerungen zwischen Anforderung eines Blocks und dessen Empfang in eine Protokolldatei.

3.5.3.1 Erzeugen einer Zufallsnachricht

Per Zufallsgenerator wurde eine zufällige Nachricht (aus Zeichen mit ASCII-Werten zwischen 32 und 94) erzeugt; Zeichen 95 wird ausgelassen, um die Anfangs- und Endblöcke eindeutig zu erhalten.

```
#!/usr/bin/python

import random
random.seed()
i=0
s=""
while i < 128:
  # Waehle Zeichen zwischen ASCII 32 und 94 (nicht: 95, wg. Begrenzung)
  s = s+chr( random.randrange(32,95) )
  i+=1

f = open("zufall.dat","w")
f.write(s)
f.close()
```

Die Zufallsnachricht für die folgenden Tests wurde durch Aufruf von `zufall.py` erzeugt und in `zufall.dat` gespeichert.

```
> ./zufall.py
> ls -l zufall.dat
-rw-r--r--    1 esser    users         128 2004-11-08 00:23 zufall.dat
> cat zufall.dat
O@ .,NN'.:9I1]D4<S';]&S>ASUL,K?L&,KO5'\SO<TOY"W5SGR6]D*;P+9&I@&AO2 =UA*VH>!]R%M.
GP8Q)P@..ZO-SMU:$HT/Z7/!E$ *Y%.3HM^2#**5QS"*21(N
```

3.5.3.2 Nachricht in die FIFO-Liste eintragen

```
> ccctrl add 192.168.1.1
OK
> ccctrl msgfile 192.168.1.1 /home/.../zufall.dat
OK
> ccctrl allstat
Info for IP address 192.168.1.1
Chars sent:  0,0
Msg length:  132
Start Trans: Mon Nov  8 00:50:27 2004
Last access: --
Message:      _ O@ .,NN'.:9I1]D4<S';]&S>ASUL,K?L&,KO5'\SO<TOY"W5SGR6]D*;P+9&I@&AO
2 =UA*VH>!]R%M̄.GP8Q)P@..ZO-SMU:$HT/Z7/!E$ *Y%.3HM^2#**5QS"*21(N _
```

Die 132 Zeichen (128 plus vier Zeichen für die Markierung von Anfang und Ende) speichert der Daemon intern als $132 * 6 = 792$ Bits. Da mit jedem 4096-Byte-Block ein Bit übertragen wird, muss die vom Apache-Server geladene Datei mindestens $4096 * 792 = 3244032$ Byte bzw. 3,09375 MByte groß sein.

3.5.3.3 Starten des Client-Proxys und Datenübertragung

Auf der Empfängerseite wird der Client-Proxy in seinem Verzeichnis durch Eingabe von

```
./ccproxy &
```

gestartet – er schreibt Messwerte in die Datei `myproxy.log`.

Die Testdatei wird dann mit

```
export http_proxy="http://localhost:21000"
time wget --proxy=on http://server:8080/random.data
```

heruntergeladen. Durch das vorangestellte `time`-Kommando wird nach dem Herunterladen die Gesamtübertragungszeit ausgegeben.

Um die jeweils letzte Datenübertragung auszuwerten, betrachtet man die vorletzte Zeile der Protokolldatei. Diese wird von einem Skript vorverarbeitet und dann an das Analyseprogramm `ccanalyse` übergeben.

Details zur Anwendung der Hilfsprogramme liefert Anhang B.

3.6 Statistische Auswertung

Bei ersten Tests wurde die Verzögerung so groß gewählt, dass sich verzögerte und nicht verzögerte Blöcke sehr leicht auseinander halten ließen. Für kleinere Verzögerungszeiten empfiehlt sich ein statistischer Ansatz zur Auswertung der Übertragungszeiten.

Bei der Übertragung werden die Zeiten zwischen Ausführen des `recv`-Befehls und dem Empfang der Daten in eine Protokolldatei geschrieben. Im Folgenden seien $d_0, d_1, \ldots$ diese Zeiten.

Ist t_0 die durchschnittliche Zeit, die für eine unverzögerte Übertragung benötigt wird, und v die künstliche Verzögerung, so kann man erwarten, dass die Sequenz aller gemessenen Übertragungszeiten zwei „Häufungspunkte" t_0 und $t_0 + v$ besitzt (siehe Abbildung 3.2).

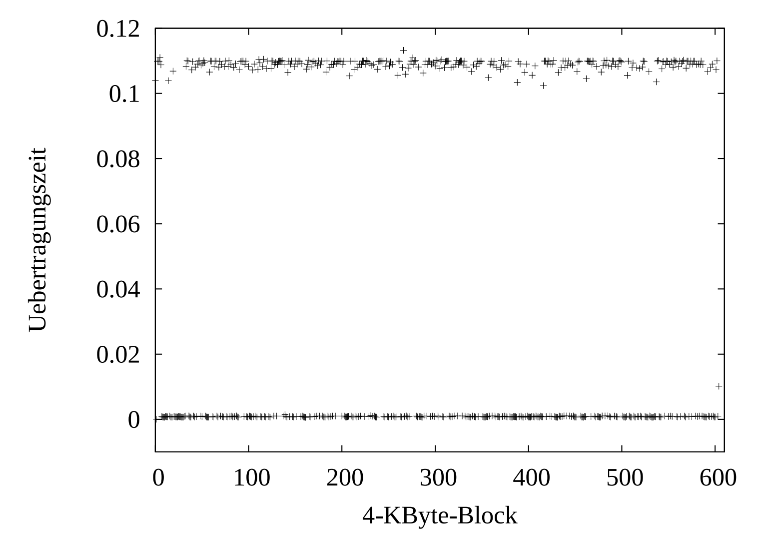


Abbildung 3.2: Häufungspunkte $t_0 \approx 0$ s und $t_1 \approx 0,11$ s bei lokalem Test mit Verzögerung $v = 0,1$ s.

Aus der Abbildung ist ersichtlich, dass im Beispielszenario auch eine Verzögerung von $v = 0,03$ s ausreichen würde, um die beiden Blöcke deutlich voneinander zu trennen.

Abbildung 3.3 zeigt die Ergebnisse für $v = 0,03$ s. In beiden Beispielen kann man erkennen, dass die unverzögerten Zeiten nahe bei 0 s liegen, während die verzögerten Zeiten sich bei $v + 0,01$ s häufen, aber auch teilweise knapp über v liegen. (t_0 ist in Relation zu großen Verzögerungen v annähernd 0.)

Diese zusätzliche Verzögerung von bis zu $0,01$ s dürfte durch die Aufrufkosten der Funktion `better_sleep` begründet sein. Es wurde ein Versuch unternommen, `better_sleep()` zu ersetzen: Die Funktion `usleep()` aus der GNU-C-Bibliothek erwartet ein Argument in Mikrosekunden. Ein Austausch von `better_sleep(0.03)` gegen `usleep(30000)` änderte aber nichts an den Abweichungen.

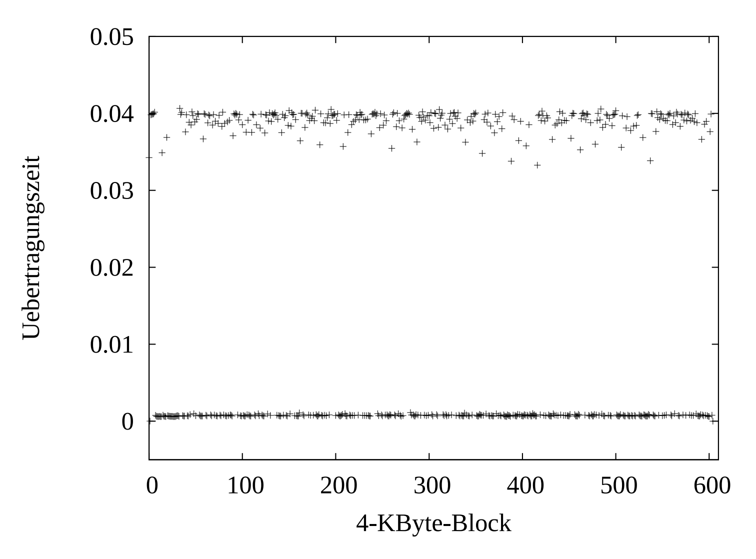


Abbildung 3.3: Häufungspunkte $t_0 \approx 0$ s und $t_1 \approx 0,04$ s bei lokalem Test mit Verzögerung $v = 0,03$ s.

Wenn t_0 und v auf Empfängerseite nicht bekannt sind, kann man in der Sequenz nach „Häufungspunkten" suchen. (Hiermit sind nicht die in der Analysis verwendeten Häufungspunkte unendlicher Folgen gemeint, da nur endliche Mengen von Messwerten betrachtet werden.) Hat man zwei Häufungspunkte $h_0 < h_1$ bestimmt, so kann man die Dekodierfunktion $D(d_i)$ wie folgt definieren:

$$D(d) := \begin{cases} 1, & \text{falls } |d - h_1| < |d - h_0| \\ 0, & \text{sonst} \end{cases}$$

Eine Übertragung wird also als verzögert interpretiert, wenn die Übertragungszeit näher am größeren Häufungspunkt als am kleineren ist.

Ziel der Optimierung ist es, in Abhängigkeit von t_0 die Verzögerung v so klein zu wählen, dass die Folge (d_i) gerade noch zwei Häufungspunkte besitzt.

3.6.1 Trennung der Werte

Definition

Sei $(d_i)_{i=0}^n$ eine endliche Folge von Werten $d_i \in \mathbb{R}$ (für alle i). Dann heißt (d_i) **trennbar**, falls die folgenden Bedingungen erfüllt sind:

1. Es gibt zwei Mengen $A, B \subset \mathbb{R}$ mit $A \cap B = \varnothing$ und $A \cup B = \{d_i | i = 0, \ldots, n\}$.
2. Mit den Mittelwerten

$$a := (\sum_{x \in A} x)/|A| = \text{avg } A, \quad b := (\sum_{x \in B} x)/|B| = \text{avg } B$$

gibt es ein $0 < \varepsilon < \frac{|b-a|}{3}$, so dass $A \subseteq U_\varepsilon(a)$ und $B \subseteq U_\varepsilon(b)$.
(Für $x, \varepsilon \in \mathbb{R}$ ist $U_\varepsilon(x) = \{y \in \mathbb{R} : |x - y| < \varepsilon\}$ die ε-Umgebung von x.)

(A, B, ε) heißt **Trennung** der (d_i).

Bemerkung

Es folgt aus Teil 2 direkt:

$$\max A < \min B \quad \text{und} \quad |\min B - \max A| > \varepsilon$$

Denn: $\max A \in A \subseteq U_\varepsilon(a)$, d. h., $|\max A - a| < \varepsilon$. Da $a < \max A$ ist, gilt $\max A - a < \varepsilon$. Analog gilt $b - \min B < \varepsilon$. Daraus folgt:

$\min B - \max A = \min B - b + b - \max A + a - a = (\min B - b) + (a - \max A) + (b - a) > (-\varepsilon) + (-\varepsilon) + 3\varepsilon = \varepsilon$.

3.6.2 Erkennung der verdeckten Nachricht

Um die durch Verzögerungen übertragene verdeckte Nachricht zu erkennen, werden die Übertragungszeiten gemessen und anschließend analysiert. Das bedeutet, dass zu den Verzögerungszeiten (d_i) eine Trennung (A, B, ε) gefunden werden muss.

Wenn die künstliche Verzögerung v sehr groß und dem Empfänger bekannt ist, ist diese Aufgabe leicht: Werte $\geq v$ entsprechen Einsen, Werte $< v$ Nullen.

Ist die Verzögerung unbekannt, muss die Analyse die typischen Zeiten für verzögerte und nicht verzögerte Übertragung selbst ermitteln. Der im Folgenden beschriebene Algorithmus geht davon aus, dass die verdeckte Botschaft gleichverteilt aus Nullen und Einsen besteht. Da die im Trägerkanal übertragene Datei eventuell größer als „nötig“ ist, muss man damit rechnen, dass am Ende der verdeckten Botschaft weitere Nullen „empfangen“ werden, die nicht Teil der Nachricht sind: Der Empfang eines unverzögerten Pakets hat immer die beiden möglichen Interpretationen „Sender hat eine 0 geschickt“ und „Sender kommuniziert derzeit nicht“. Diesen Bereich kann man wie folgt erkennen:

Es seien $(d_0, \ldots, d_n)$ die endliche Folge aller protokollierten Übertragungszeiten und avg$(d_k, d_{k+1}, \ldots, d_{k+m}) := (\sum_{i=k}^{k+m} d_i)/(m+1)$ die Mittel der Übertragungszeiten d_k bis d_{k+m}.

Zunächst wird geprüft, ob die Nachricht am Ende überflüssige Nullen enthält. Ein solcher Bereich $(d_k, \ldots, d_n)$ (für ein $k \in \mathbb{N}$) zeichnet sich durch geringe Abweichungen aus. Die Standardabweichung für ein Endstück $(d_k, \ldots, d_n)$ kann auf Basis der vorhandenen Daten wie folgt berechnet werden:

$$\overline{d}_{k\ldots n} := \frac{1}{n-k+1} \sum_{i=k}^{n} d_i$$

$$\sigma_{k\ldots n} := \sqrt{\frac{1}{n-k+1} \sum_{i=k}^{n} (d_i - \overline{d}_{k\ldots n})^2}$$

Der erste Wert, $\overline{d}_{k\ldots n}$, ist die durchschnittliche Übertragungszeit von der k-ten bis zur n-ten (letzten) Position; daraus berechnet sich der zweite Wert, die Standardabweichung. (Mit $\overline{d} := \overline{d}_{0\ldots n}$ und $\sigma := \sigma_{0\ldots n}$ erhält man die entsprechenden Werte für die Gesamtnachricht.)

Ist k die Position, ab der nur noch Nullen empfangen wurden, müssen die Abweichungen $\sigma_{k\ldots n}, \sigma_{k+1\ldots n}, \ldots$ niedrig sein. Durch Untersuchung der Folge $(\sigma_{k\ldots n})_{k=1}^{n}$ kann man das Ende der Nachricht erkennen.

Ist damit die tatsächliche Länge der Nachricht bestimmt, wird das inhaltslose Reststück abgeschnitten. Im Folgenden sei $(d_0, \ldots, d_n)$ die vollständige Nachricht ohne Reststück.

Für die verbliebene Nachricht geht man nun von einer Gleichverteilung der Nullen und Einsen aus, damit sollten annähernd gleich viele verzögerte wie nicht verzögerte Werte vorliegen, so dass man eine Trennung der (d_i) findet, indem man zunächst mit

$$A_0 := \{d_i | d_i < \text{med } (d_j)_{j=0}^{n}\}$$

$$B_0 := \{d_i | d_i \geq \text{med } (d_j)_{j=0}^{n}\}$$

die Werte oberhalb und unterhalb des Medians betrachtet und anschließend A und B bildet, indem man Elemente aus A_0 und B_0, die sich näher am Mittelwert der jeweils anderen Menge als am Mittelwert der eigenen befinden, verschiebt:

$$A2B := \{d \in A_0 : |d - \text{avg } A_0| > |d - \text{avg } B_0|\}$$

$$B2A := \{d \in B_0 : |d - \text{avg } B_0| > |d - \text{avg } A_0|\}$$

$$A := (A_0 \setminus A2B) \cup B2A$$

$$B := (B_0 \setminus B2A) \cup A2B$$

Mit $h_0 := \text{avg}A$ und $h_1 := \text{avg}B$ wird dann die Dekodierfunktion D beschrieben, es gilt dann auch

$$D(d) := \begin{cases} 1, & \text{falls } d \in B \\ 0, & \text{falls } d \in A \end{cases}$$

3.6.3 Das Analyseprogramm

Im Rahmen dieser Arbeit wurde ein Analyseprogramm entwickelt, das einige der beschriebenen Techniken umsetzt. Auf eine Analyse des Nachrichtenendes wurde verzichtet, da in der Auswertung die Länge der Originalnachricht bekannt war und Verzögerungszeiten ignoriert wurden, die nach dem Empfang der vollständigen Nachricht gemessen wurden.

Das in Python geschriebene Programm `ccanalyse` liest die Liste der vom Client-Proxy gemessenen und protokollierten Verzögerungen ein. Diese wird in zwei Blöcke aufteilt, wobei als Trenner der Median der Verzögerungen gewählt wird: Das Analyseprogramm setzt hier voraus, dass annähernd gleich viele verzögerte und unverzögerte Pakete gesendet wurden.

Wie im vorangehenden Unterkapitel beschrieben, erfolgt nach der Mediantrennung noch eine Anpassung: Werte, die näher am Mittelwert der jeweils anderen Menge liegen, werden dorthin verschoben. Das leistet in `ccanalyse` die Funktion `fixvals()`:

```
def fixvals(A,B):
   a = mittelwert(A)
   b = mittelwert(B)
   A2B = []   # Werte, die von A nach B wandern sollen
   B2A = []   # Werte, die von B nach A wandern sollen

   for v in A+B:
      if v in A and abs(v-a) > abs (v-b): A2B.append(v)
      if v in B and abs(v-b) > abs (v-a): B2A.append(v)

   for v in A2B: A.remove(v); B.append(v)
   for v in B2A: B.remove(v); A.append(v)

   if max(A) > min(B): print "Fehler␣in␣fixvals()"

   return (A,B)
```

Der Rest der Analyse besteht aus der Dekodierung der Messwerte anhand der gefundenen Trennung; `ccanalyse` berechnet zudem die Fehlerrate und gibt einige statistische Informationen aus.

Eine vollständige und kommentierte Version des Analyseprogramms `ccanalyse` ist in Anhang C.3 ab Seite 123 zu finden.

3.7 Ergänzung der Fehlerkorrektur

Da im Internet-Szenario selbst bei hohen Verzögerungszeiten (bis zu einer halben Sekunde) keine vollständig fehlerfreie Übertragung möglich war, wurde in die Kommunikation ein Fehlerkorrekturverfahren eingebaut, das jedes (in 6 Bit kodierte) übertragene Zeichen um vier Paritätsbits ergänzt. Die theoretischen Grundlagen und die Konstruktion der Paritätsvektoren sind in Kapitel 2.7.2 (ab Seite 27) beschrieben.

Als Paritätsvektoren dienten (0, 0, 0, 0, 1, 1), (0, 1, 1, 1, 0, 0), (1, 0, 1, 1, 0, 1) und (1, 1, 0, 1, 1, 0), d. h., aus einem 6-Bit-Eingabewort $(x_0, \ldots, x_5)$ erzeugt die Funktion $h(x_0, \ldots, x_5) := (y_0, \ldots, y_9)$ mit

$$y_0 = x_0, \ldots, y_5 = x_5$$
$$y_6 = x_4 + x_5$$
$$y_7 = x_1 + x_2 + x_3$$
$$y_8 = x_0 + x_2 + x_3 + x_5$$
$$y_9 = x_0 + x_1 + x_3 + x_4$$

ein 10-Bit-Wort. Dabei ist $+$ die binäre Addition über $\{0, 1\}$ mit $0 + 0 = 1 + 1 = 0$ und $0 + 1 = 1 + 0 = 1$.

In Matrixschreibweise kann man h wie folgt darstellen:

$$y = xH, \text{ wobei } H = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ 0 & 1 & 0 & 0 & 0 & 0 & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ 0 & 0 & 1 & 0 & 0 & 0 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 0 & 0 & 0 & 1 & 0 & 0 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & 0 & 1 & 0 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 1 & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \end{pmatrix}$$

Der linke (nicht fett hervorgehobene) Teil dieser Matrix ist die Matrixrepräsentation der Identitätsfunktion, da der Ergebnisvektor in den ersten sechs Komponenten mit dem Argument identisch ist. Die letzten vier Spalten enthalten (transponiert) die vier Paritätsvektoren.

Der Rang dieser Matrix (also die Anzahl der über $\{0, 1\}$ linear unabhängigen Zeilen und/oder Spalten) ist 6, so dass der Bildbereich Bild(h) genauso viele Elemente enthält wie $\{0, 1\}^6$ (und zwar $2^6 = 64$).

Da die Hamming-Distanz von Bild(h) den Wert 3 hat, kann auf der Empfängerseite ein Bitfehler pro übertragenem 6-Bit-Zeichen korrigiert werden.

3.7.1 Implementierung der Fehlerkorrektur

Um die Fehlerkorrektur in die bestehenden Quelltexte zu integrieren, waren Änderungen in zwei Python-Programmen notwendig:

- Auf Server-Seite wurde der Kontroll-Daemon angepasst. Über eine Variable `ERROR_CORRECTION` wird festgelegt, ob Fehlerkorrektur verwendet wird oder nicht – so lässt sich der Daemon wahlweise im normalen oder im Fehlerkorrekturmodus betreiben.

  Ist Fehlerkorrektur aktiviert, wird dem Bitkodierer (der ASCII-Zeichenketten in Folgen aus Nullen und Einsen konvertiert) eine Funktion zur Fehlerkorrektur nachgeschaltet.

- Auf Client-Seite wird nur das Analyse-Tool um einen Dekodierer ergänzt, der 10-Bit-Folgen aus Nullen und Einsen in die erwarteten 6-Bit-Folgen zurückwandelt.

Der Apache-Server und der Proxy-Server auf der Client-Seite bleiben unverändert.

3.8 Zusammenfassung

Dieses Kapitel hat nach einem einleitenden Trivialbeispiel die Implementierungsarbeiten vorgestellt, die im Rahmen dieser Arbeit erfolgten.

- Im Quellcode des Web-Servers Apache (in Version 1.3.31) wurden zwei Funktionen, die für den Versand von Datenpaketen an einen Client zuständig sind, so erweitert, dass vor jedem Versand eines Datenblocks ein Daemon-Prozess kontaktiert und gefragt wird, ob dieser Block verzögert werden soll. Die Blockgröße wurde in beiden Funktionen auf 4096 Byte gesetzt.
- Es wurde ein Daemon-Programm entwickelt, das mit dem Apache-Server und mit einem Kontrollprogramm kommuniziert. Es verwaltet ein *Dictionary* von IP-Adressen (Kommunikationspartnern) und zugehörigen Statusinformationen einschließlich der an diese Adressen verdeckt zu sendenden Nachrichten.
- Ein Kontrollprogramm wird verwendet, um dem Daemon-Prozess mitzuteilen, mit welchen Rechnern verdeckte Kommunikation stattfinden soll. Auch die Nachrichten selbst werden auf diesem Weg an den Daemon übermittelt.
- Für die Client-Seite wurde ein transparenter HTTP-Proxy entwickelt, der Anfragen eines Clients an einen Server weiterleitet und dabei die Übertragungszeiten einzelner Blöcke misst und protokolliert.
- Ein Analyseprogramm wertet (mit Hilfe einiger kleinerer Hilfsprogramme) die gemessenen Übertragungszeiten aus und bestimmt so unter anderem die Fehlerrate.

Alle Komponenten arbeiten einwandfrei zusammen.

Das folgende Kapitel stellt die Messergebnisse aus mehreren Testreihen vor, die lokal, im lokalen Netzwerk und über eine Internet-Verbindung erfolgten. Auswirkungen der Fehlerkorrektur werden beschrieben, und die Messwerte werden mit den theoretischen Ergebnissen zur erwarteten Kapazität verglichen.

Kapitel 4

Analyse

Dieses Kapitel stellt die Messwerte vor, die beim Test der Implementierung aufgezeichnet wurden. In der Diskussion werden die Ergebnisse mit der theoretisch zu erwartenden Kapazität verglichen, und die Frage nach der Beobachtbarkeit wird beantwortet.

4.1 Einleitung

Im Anschluss an die Implementierung wurde ihre Leistungsfähigkeit untersucht: Interessant waren dabei die folgenden Fragen:

- Welche Datenrate (relativ zum Trägerkanal) lässt sich auf dem verdeckten Kanal erreichen?
- Wie zuverlässig ist der verdeckte Kanal?
- Wie beeinflussen die Qualität der Netzwerkverbindung zwischen Client und Server und die Auslastung des Servers die Datenrate?
- Wurde das Ziel erreicht, dass die verdeckte Kommunikation sich durch Kontrollen mit Paket-Sniffern nicht feststellen lässt?

Für die Untersuchungen wurden vier Testszenarien mit unterschiedlichen Netzwerkbedingungen geschaffen:

1. **Lokaler Test:** Alle Komponenten laufen auf dem gleichen Rechner.
2. **Test im lokalen Netz:** Alle Komponenten laufen auf zwei Rechnern, die dem gleichen lokalen Netzwerk angehören und über 100 MBit Ethernet miteinander verbunden sind.
3. **Internet, lastfrei:** Server und Kontrollprogramm laufen auf einem per DSL-Internet-Zugang erreichbaren Root-Server (der bei 1&1 gehostet wird). Der Server ist nicht durch sonstige Anfragen belastet.
4. **Internet, unter Last:** Situation wie unter 3), aber mit gleichzeitiger Belastung des Servers durch parallele Downloads von anderen Rechnern im Internet aus.

In allen vier Szenarien wurden Testdaten für unterschiedliche Verzögerungszeiten erhoben. Dabei wurde jeweils die Fehlerrate ermittelt. Es war zu erwarten, dass die besten Ergebnisse beim lokalen Test, die schlechtesten beim Internet-Test unter Last erreicht werden.

Anschließend wurde getestet, ob der Durchsatz erhöht werden konnte, indem bei niedriger (und zu Fehlern führender) Verzögerung ein Fehlerkorrekturverfahren verwendet wurde.

Das nächste Unterkapitel 4.2 beschreibt die grundsätzliche Vorgehensweise. Die folgenden Unterkapitel stellen die Ergebnisse in den vier Szenarien

- lokal: Kapitel 4.3 (ab Seite 67),
- lokales Netz: Kapitel 4.4 (ab Seite 69),
- Internet ohne Last: Kapitel 4.5 (ab Seite 74),
- Internet unter Last: Kapitel 4.6 (ab Seite 82)

vor, wobei für die Internet-Tests auch die Veränderungen durch den Einsatz der Fehlerkorrektur besprochen werden.

Kapitel 4.7 (ab Seite 83) behandelt die Beobachtbarkeit des verdeckten Kanals, im folgenden Kapitel 4.8 (ab Seite 84) wird die gemessene Kapazität mit der theoretisch zu erwartenden verglichen.

Das Kapitel endet mit einer Darstellung der Beschränkungen (Kapitel 4.9 ab Seite 91) und einer Zusammenfassung (Seite 92).

4.2 Vorgehensweise

Für alle Tests wurde die in Kapitel 3.5.3.1 (Seite 56) beschriebene zufällige Nachricht verwendet. Auf dem regulären Kanal wurde eine ausreichend große Datei übertragen, so dass die gesamte verdeckte Nachricht während einer einzigen HTTP-Anfrage übermittelt werden konnte. Diese 5 MByte große Datei für das Trägermedium wurde mit

```
dd if=/dev/urandom of=/var/apache/httpd_docs/test/random.data bs=1024 count=5120
```

mit Zufallswerten gefüllt erzeugt. Ihr Inhalt spielt allerdings keine Rolle, da das Verhalten des modifizierten Apache-Servers nicht vom Inhalt der Daten auf dem regulären Kanal beeinflusst wird.

Für die unterschiedlichen Verzögerungszeiten waren jeweils die Anpassung der Präprozessor-Variable `AP_CC_SLEEPTIME` in der Datei `include/covert_channel.h` und eine anschließende Neukompilierung des Apache-Servers notwendig.

Vor jedem Testlauf wurde die Warteschlange für die Client-IP-Adresse neu initialisiert, d. h., eventuelle Reste der verdeckten Nachricht wurden durch Löschen des

Warteschlangeneintrags entfernt, die Warteschlange neu angelegt und mit der Testnachricht aus Kapitel 3.5.3.1 gefüllt.

Danach wurde jeweils mit `wget` und unter Verwendung des CC-Proxys die Datei `random.data` vom Apache-Server heruntergeladen, es folgte dann eine Auswertung der vom Proxy in eine Protokolldatei geschriebenen Verzögerungszeiten.

Im Folgenden bezeichnet v stets die Verzögerungszeit, die für den jeweiligen Test im Apache-Server als Wartezeit angegeben wurde.

4.3 Lokaler Test

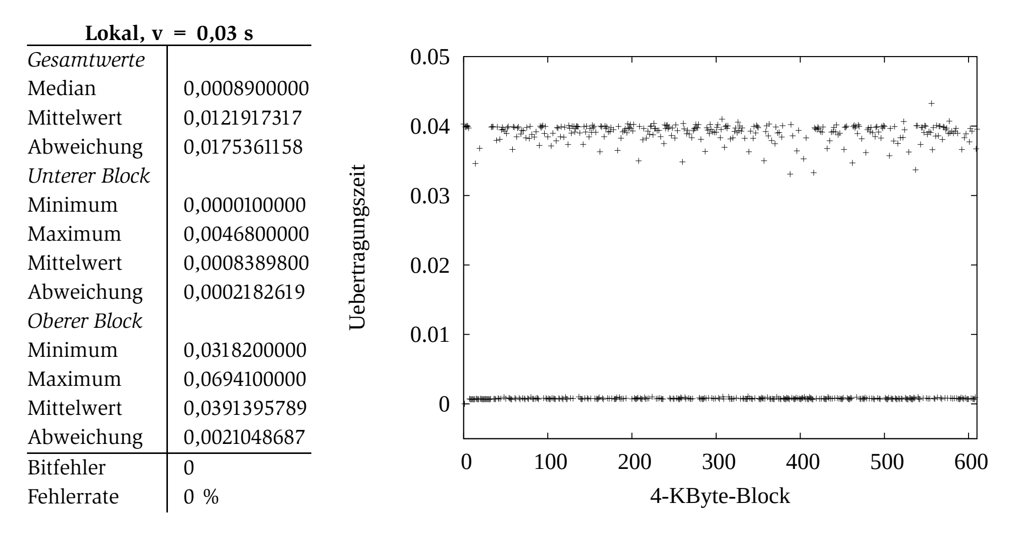
Der lokale Test ist durch die minimale Laufzeit von IP-Paketen zwischen Sender und Empfänger gekennzeichnet: Die Daten verlassen den Rechner nicht.

Auch sonstige Beeinflussungen des Testrechners, etwa durch hohe Systemlast, wurden ausgeschlossen; der vom Dienstprogramm `top` angezeigte Wert *load average* lag nahe bei 0.

4.3.1 Messergebnisse

0,03 Sekunden: Für $v = 0,03$ s wurden die in Abbildung 4.1 (Seite 67) dargestellten Werte gemessen. Die Trennung der beiden Bereiche ist in der Grafik deutlich erkennbar. Die meisten Werte des oberen Bereichs liegen um 0,04 s – nicht beim erwarteten Wert von knapp oberhalb der Verzögerung (0,03 s). Die Übertragung war fehlerfrei.

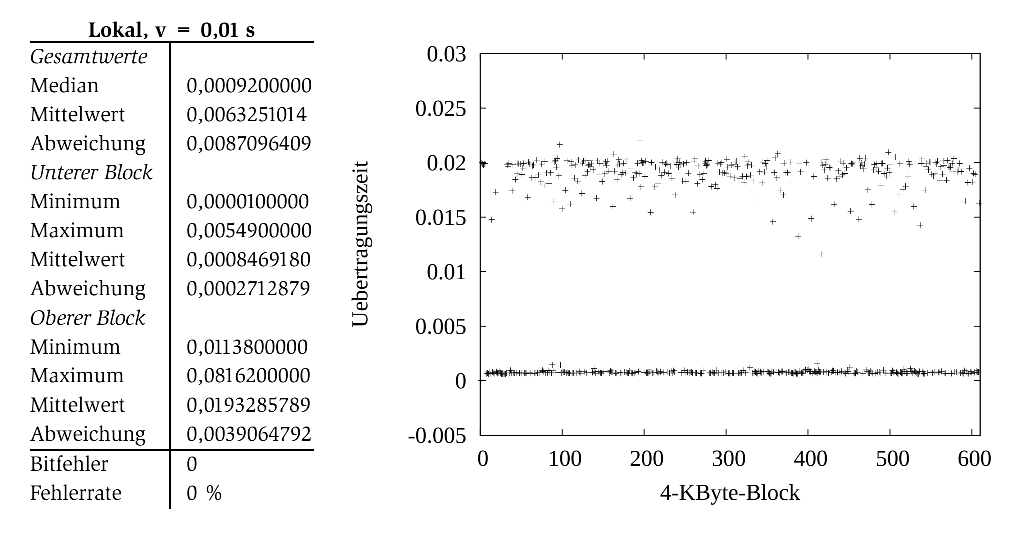
Aus dem großen Abstand ergibt sich, dass eine Verringerung der Verzögerung problemlos möglich sein sollte.



Lokal, v = 0,03 s	
Gesamtwerte	
Median	0,0008900000
Mittelwert	0,0121917317
Abweichung	0,0175361158
Unterer Block	
Minimum	0,0000100000
Maximum	0,0046800000
Mittelwert	0,0008389800
Abweichung	0,0002182619
Oberer Block	
Minimum	0,0318200000
Maximum	0,0694100000
Mittelwert	0,0391395789
Abweichung	0,0021048687
Bitfehler	0
Fehlerrate	0 %

Abbildung 4.1: Messwerte und Grafik für den lokalen Test mit $v = 0,03$ s.

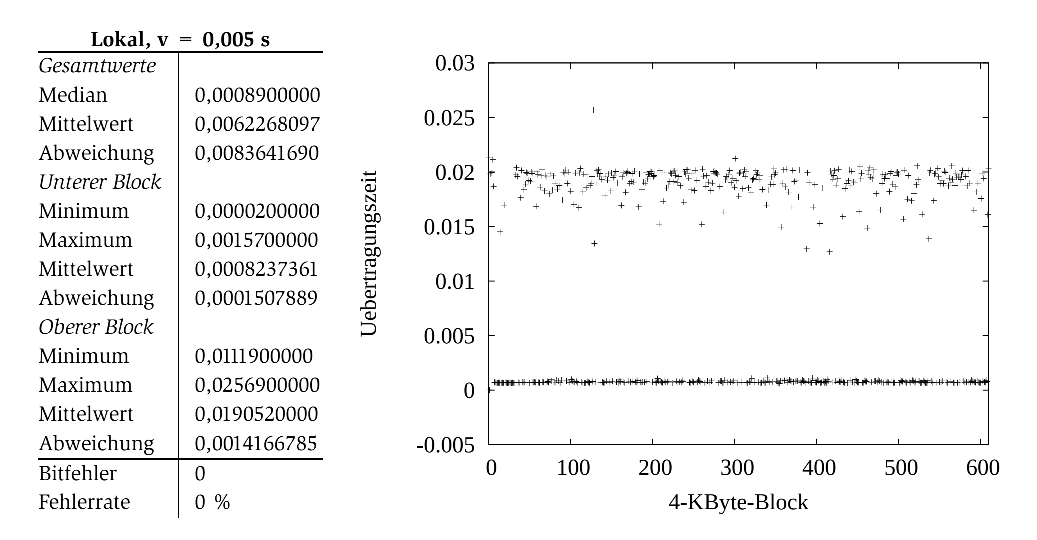
0,01 Sekunden: Für $v = 0,01$ s wurden die in Abbildung 4.2 dargestellten Werte gemessen. Wie erwartet, blieb der Abstand zwischen den beiden Bereichen ausreichend groß: Die Übertragung war auch mit 0,01 s Verzögerung noch fehlerfrei.



Lokal, v = 0,01 s	
Gesamtwerte	
Median	0,0009200000
Mittelwert	0,0063251014
Abweichung	0,0087096409
Unterer Block	
Minimum	0,0000100000
Maximum	0,0054900000
Mittelwert	0,0008469180
Abweichung	0,0002712879
Oberer Block	
Minimum	0,0113800000
Maximum	0,0816200000
Mittelwert	0,0193285789
Abweichung	0,0039064792
Bitfehler	0
Fehlerrate	0 %

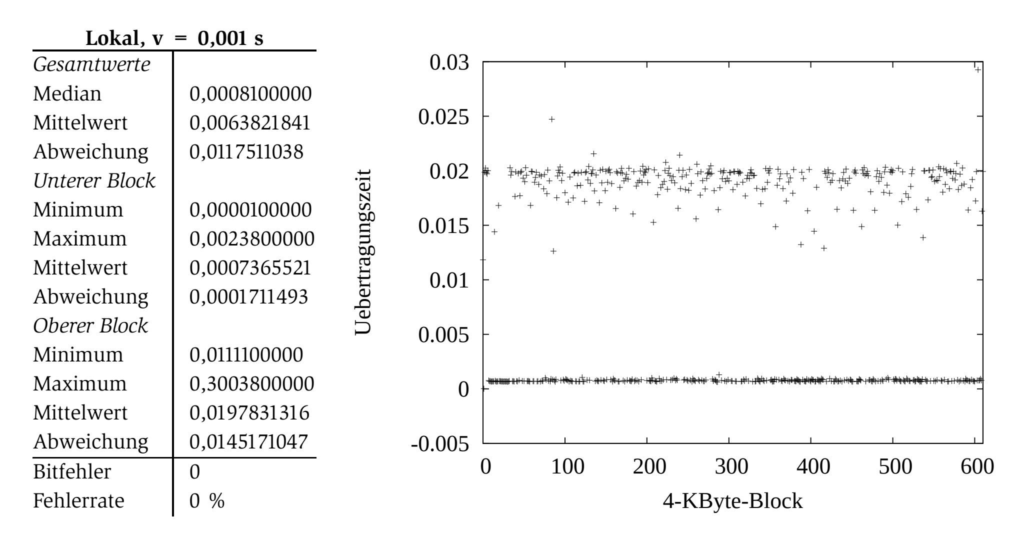
Abbildung 4.2: Messwerte und Grafik für den lokalen Test mit $v = 0,01$ s.

0,005 und 0,001 Sekunden: Die Tests mit $v = 0,005$ s (Abbildung 4.3) und $v = 0,001$ s (Abbildung 4.4 auf Seite 69) lieferten nicht das gewünschte Ergebnis, nämlich eine weitere Annäherung der beiden Blöcke – wie die beiden Abbildungen zeigen, ändert sich im Vergleich zu $v = 0,01$ s kaum etwas an der Verteilung der Punkte im oberen Block.



Lokal, v = 0,005 s	
Gesamtwerte	
Median	0,0008900000
Mittelwert	0,0062268097
Abweichung	0,0083641690
Unterer Block	
Minimum	0,0000200000
Maximum	0,0015700000
Mittelwert	0,0008237361
Abweichung	0,0001507889
Oberer Block	
Minimum	0,0111900000
Maximum	0,0256900000
Mittelwert	0,0190520000
Abweichung	0,0014166785
Bitfehler	0
Fehlerrate	0 %

Abbildung 4.3: Messwerte und Grafik für den lokalen Test mit $v = 0,005$ s.



Lokal, v = 0,001 s	
Gesamtwerte	
Median	0,0008100000
Mittelwert	0,0063821841
Abweichung	0,0117511038
Unterer Block	
Minimum	0,0000100000
Maximum	0,0023800000
Mittelwert	0,0007365521
Abweichung	0,0001711493
Oberer Block	
Minimum	0,0111100000
Maximum	0,3003800000
Mittelwert	0,0197831316
Abweichung	0,0145171047
Bitfehler	0
Fehlerrate	0 %

Abbildung 4.4: Messwerte und Grafik für den lokalen Test mit $v = 0,001$ s.

4.4 Test im lokalen Netz

Die Tests im lokalen Netz, also mit zwei über 100 MBit Ethernet verbundenen Rechnern, erwiesen sich zunächst als problematisch: Bei den Testläufen entstand eine größere Menge Messdaten als üblich, und ihre Dekodierung führte zu keinen sinnvoll interpretierbaren Ergebnissen – das konnte selbst durch Heraufsetzen der Verzögerung auf eine halbe Sekunde nicht verbessert werden.

Eine Analyse der in jedem Schritt übertragenen Datenpakete führte dann zur Lösung: Während im rein lokalen Fall bei wirklich jedem Aufruf der Funktion `recv()` auf Empfängerseite 4096 Bytes gelesen wurden, kamen im lokalen Netz auch häufig kleinere Pakete an. Das lokale Netzwerk-Interface `lo` (Loopback, 127.0.0.1) verwendet eine MTU von 16436, während die Ethernet-Karte auf 1500 konfiguriert ist.

Ein Test mit Ethereal [eth] bestätigte die Vermutung: Bei lokaler Übertragung (über das Netzwerk-Interface `lo`) wurden 4162 Byte große Pakete übertragen, die jeweils ein 4096 Byte großes HTTP-Datenpaket enthielten.

Der gleiche Datentransfer über das lokale Netzwerk führte zu einer Aufteilung (Fragmentierung) in drei TCP-Pakete zu 1500, 1500 und 1252 Byte mit enthaltenen HTTP-Paketgrößen von 1448, 1448 und 1200 (in der Summe also 4096) Byte (Abbildung 4.5 auf Seite 70). Die Fragmentierung fand auf TCP-Ebene statt, wie am nicht gesetzten IP-*Flag* `More Fragments` zu erkennen war (vgl. Kapitel 2.8.1 auf Seite 31).

Dadurch wurde die bisherige Analysemethode, die einfach die von `recv()` benötigte Zeit maß, unbrauchbar.

Die Lösung bestand darin, im Proxy die empfangenen Bytes zu zählen und die Übertragungszeiten zu addieren, bis ein gesamtes 4096-Byte-Päckchen empfangen wurde – nach dieser Änderung war die Anzahl der Messwerte wieder korrekt.

In der ursprünglichen Version des Proxys sah der Block, der die Daten holt und weiterleitet, wie folgt aus:

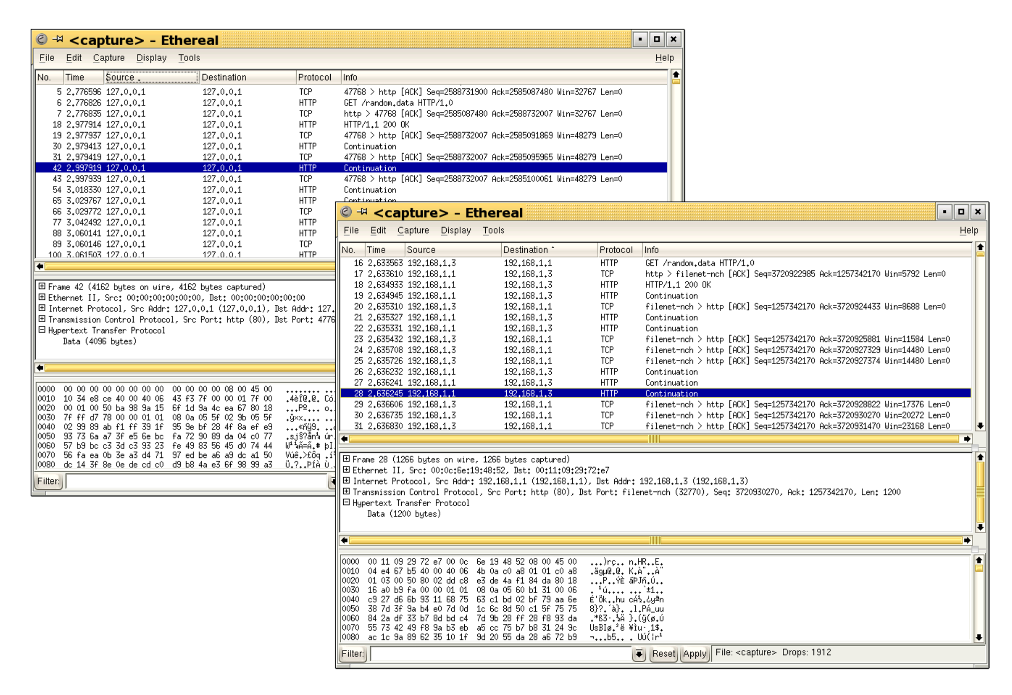


Abbildung 4.5: Beobachtung des Netzwerkverkehrs mit `ethereal` – links: keine Fragmentierung bei lokaler Übertragung; rechts: Aufteilung in drei Pakete im lokalen Netzwerk.

```
[...]
    # Blockweise holen und zustellen
    done=0
    psize=4096                        # Paketgröße für recv(): 4 KByte
    times=[]                          # Hier die Wartezeiten speichern
    while not done:
        t0=time()
        result=s_web.recv(psize)
        t1=time()
        times.append(t1-t0)           # Verzögerung speichern
        c.send(result)
        # stdout.write(".")
        print len(result),
        if result=="": # or len(result)<psize: done=1
            done=1
    s_web.close()
[...]
```

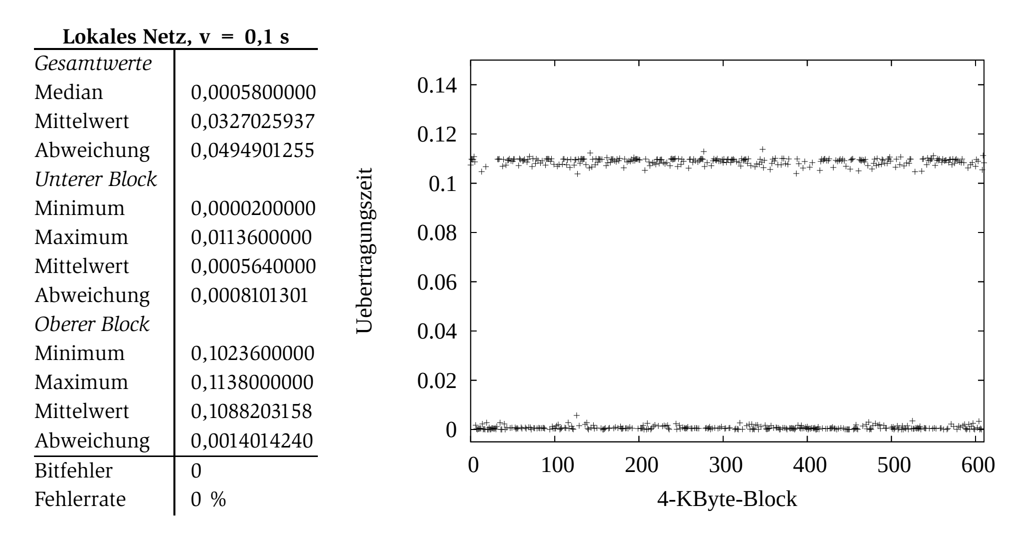
Die modifizierte Fassung wurde bereits in Kapitel 3.4.1 auf Seite 44 vorgestellt.

Empfindlichkeit gegenüber Netzwerkverkehr Bei den Tests im lokalen Netzwerk fiel ein weiteres, unerwartetes Problem auf, das einige der Testläufe unbrauchbar machte: Einer der beiden am Test beteiligten Rechner diente gleichzeitig als Rechnerarbeitsplatz – hier lief ein MP3-Player, der die MP3-Dateien per NFS (*Network File System*) vom zweiten beteiligten Rechner zog. Dieser zusätzliche Netzwerkverkehr war vom Autor anfänglich als irrelevant betrachtet worden, ruinierte aber die Übertragung auf dem verdeckten Kanel – bis hin zu 50 % Fehlerquote.

Als Konsequenz wurde für die Dauer der Testläufe auf die Belastung des NFS-Servers verzichtet.

4.4.1 Messergebnisse

0,1 Sekunden: Im ersten Anlauf wurde dem lokalen Netzwerk gegenüber dem lokalen Test eine größere Verzögerung von einer zehntel Sekunde eingeräumt. Mit $v = 0,1$ s ergaben sich die Werte aus Abbildung 4.6 auf Seite 71. Bei dieser großzügig bemessenen Verzögerung war die Übertragung auf dem verdeckten Kanal fehlerfrei.



Lokales Netz, v = 0,1 s	
Gesamtwerte	
Median	0,0005800000
Mittelwert	0,0327025937
Abweichung	0,0494901255
Unterer Block	
Minimum	0,0000200000
Maximum	0,0113600000
Mittelwert	0,0005640000
Abweichung	0,0008101301
Oberer Block	
Minimum	0,1023600000
Maximum	0,1138000000
Mittelwert	0,1088203158
Abweichung	0,0014014240
Bitfehler	0
Fehlerrate	0 %

Abbildung 4.6: Messwerte und Grafik für den Test im lokalen Netz mit $v = 0,1$ s.

Auch im Fall des lokalen Netzes ergibt sich der charakteristische Effekt, dass der Großteil der Messwerte aus dem oberen Bereich ca. 0,01 s oberhalb der Verzögerung liegt.

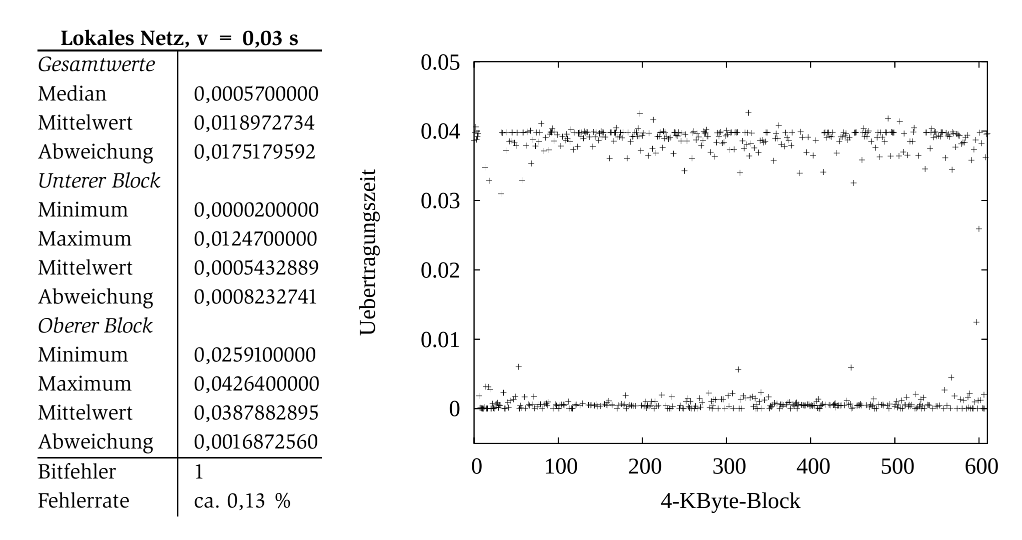
0,03 Sekunden: Schon beim Herabsetzen auf 0,03 s, die erste Verzögerung aus dem lokalen Test, kam es zu Fehlern. Abbildung 4.7 auf Seite 72 zeigt die im ersten Testlauf gemessenen Werte.

Mit einer Fehlerquote von ca. 0,13 % (ein falsches Bit von 780) liegt diese Übertragung allerdings in einem Bereich, bei dem Fehlerkorrektur möglich sein sollte – mehr dazu in Kapitel 4.5.4.

Vier Wiederholungen bei gleicher Verzögerung führten zu fehlerfreien Übertragungen, so dass sich im Schnitt 0,2 Fehler pro Testlauf bzw. eine durchschnittliche Fehlerrate von 0,03 % ergab. Diese Tests wurden unter Belastung der Netzwerkverbindung durch geringfügigen NFS-Verkehr durchgeführt.

Bei einer Wiederholung ohne Netzlast (wieder fünf Versuche) waren ebenfalls vier Übertragungen komplett fehlerfrei, und bei einer Übertragung kippte ein Bit, so dass kein qualitativer Unterschied erkennbar war.

Man muss davon ausgehen, dass bereits bei der Verzögerung von 0,03 s regelmäßig Fehler auftreten.



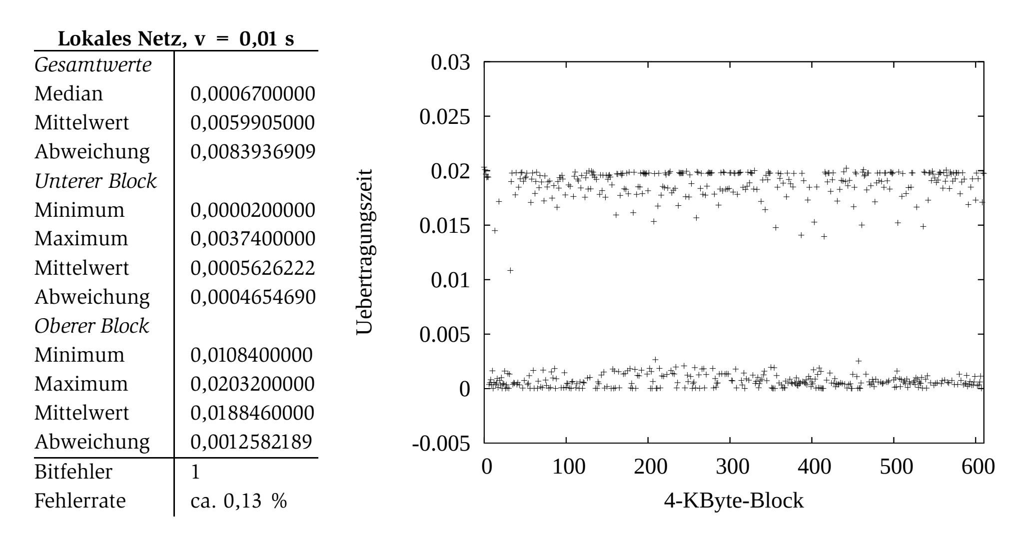
Lokales Netz, v = 0,03 s	
Gesamtwerte	
Median	0,0005700000
Mittelwert	0,0118972734
Abweichung	0,0175179592
Unterer Block	
Minimum	0,0000200000
Maximum	0,0124700000
Mittelwert	0,0005432889
Abweichung	0,0008232741
Oberer Block	
Minimum	0,0259100000
Maximum	0,0426400000
Mittelwert	0,0387882895
Abweichung	0,0016872560
Bitfehler	1
Fehlerrate	ca. 0,13 %

Abbildung 4.7: Messwerte und Grafik für den Test im lokalen Netz mit $v = 0,03$ s (fehlerbehafteter Versuch).

0,01 Sekunden: Ein weiteres Herabsetzen der Verzögerung auf 0,01 Sekunden erhöht das Fehlerrisiko, im Test wurde allerdings die gleiche durchschnittliche Fehlerrate wie bei 0,03 Sekunden gemessen.

Bei den ersten Tests mit geringfügig belasteter NFS-Verbindung zwischen den beiden Rechnern waren die Ergebnisse völlig unbrauchbar; die Fehlerraten lagen zwischen 15 % und 50 %. Nach dem Verringern des NFS-Verkehrs senkte sich die Fehlerrate auf 0 – 0,13 %, das entspricht keinem bzw. einem Bit-Fehler.

Abbildung 4.8 (Seite 72) zeigt die Werte aus dem ersten Versuch: Mit einer Fehlerquote von ca. 0,13 % (ein falsches Bit von 780) ist hier der Versuch sinnvoll, ein Fehlerkorrekturverfahren einzusetzen.



Lokales Netz, v = 0,01 s	
Gesamtwerte	
Median	0,0006700000
Mittelwert	0,0059905000
Abweichung	0,0083936909
Unterer Block	
Minimum	0,0000200000
Maximum	0,0037400000
Mittelwert	0,0005626222
Abweichung	0,0004654690
Oberer Block	
Minimum	0,0108400000
Maximum	0,0203200000
Mittelwert	0,0188460000
Abweichung	0,0012582189
Bitfehler	1
Fehlerrate	ca. 0,13 %

Abbildung 4.8: Messwerte und Grafik für den Test im lokalen Netz mit $v = 0,01$ s (erster Versuch).

0,005 Sekunden: Ein Herabsenken der Verzögerung auf 0,005 s änderte nichts an den Ergebnissen: Wie im rein lokalen Fall ähnelt das Verhalten bei 0,005 s dem bei 0,01 s. Auf eine Darstellung der Messergebnisse wird daher verzichtet.

4.4.2 Lokaler Test mit herabgesetzter MTU

Um die Messwerte der lokalen Tests und der Tests im lokalen Netzwerk vergleichen zu können, wurde die MTU für das lokale Netzwerk-Interface `lo` auf 1500 herabgesetzt – das ist auch die MTU, die das zur Netzwerkkarte gehörende Interface `eth0` verwendet. Mit dieser geänderten Einstellung und mit dem angepassten Proxy-Server wurden die Messungen für den lokalen Fall wiederholt.

Durch die niedrigere MTU kam es bei der Wiederholung auch lokal zur Aufteilung der Pakete – wie beim Test im lokalen Netzwerk wurden Blöcke mit 1500, 1500 und 1252 Byte übertragen, die enthaltenen HTTP-Pakete hatten wiederum die Größen 1448, 1448 und 1200 Byte.

Durch die Änderungen kam es zu höheren Fehlerquoten, Tabelle 4.1 vergleicht die Messungen mit MTU 16436 und MTU 1500.

Wie bei den ersten Tests wirkten sich Verzögerungen unter 0,01 s nicht mehr weiter aus: Für alle drei Fälle 0,01 s, 0,005 s und 0,001 s hatten die verzögerten Pakete eine gemessene Übertragungszeit um 0,02 s.

Damit scheint die Verringerung der MTU die Fehlerrate deutlich stärker zu beeinflussen als der Unterschied zwischen Kommunikation über die Netzwerkkarte und über das lokale Interface `lo`.

Dass die Fehlerraten sogar höher als beim Test im lokalen Netzwerk waren, dürfte an einer Konfigurationsänderung der Testgeräte zwischen den ersten Tests und den Nachtests liegen: Einer der ursprünglich verwendeten Testrechner war infolge eines Plattenschadens ausgefallen.

Verzögerung	Fehler bei MTU 16436	... MTU 1500	lokales Netz
0,030 s	0 %	0,077 %	0,03 %
0,010 s	0 %	0,205 %	0,03 %
0,005 s	0 %	0,308 %	0,03 %
0,001 s	0 %	0,205 %	—

Tabelle 4.1: Lokale Messwerte mit MTU 16436 und 1500; letzte Spalte: Werte im lokalen Netz.

4.5 Test im Internet ohne Last

Das Szenario mit einem Rechner im Internet (ein Root-Server bei einem der großen *Hosting*-Anbieter) ist das praxisnaheste: Angriffe, die nur ein lokales Intranet betreffen, sind im Vergleich zu Attacken von außen selten.

Aus diesem Grund wurden für diesen Test deutlich mehr Messwerte als im lokalen Fall erhoben: Für 15 Verzögerungswerte zwischen 0,05 s und 0,5 s wurden je fünf Testläufe ausgeführt.

4.5.1 Beschreibung der Test-Hardware

Für den Test wurden folgende Komponenten verwendet:

Server: Intel Celeron, 2 GHz, 256 MByte RAM, Debian Linux 3.0, 1 & 1 Rechenzentrum (Root-Server)

Client: Intel Pentium IV, 2 GHz, 512 MByte RAM, Suse Linux 9.0, DSL (768 MBit/s *up-stream*, 128 MBit/s *down-stream*)

Beide Rechner verwenden die MTU (*Maximum Transfer Unit*, siehe Kapitel 2.8.1) 1500, beim Aufbau von TCP-Verbindungen wird die MSS 1400 ausgehandelt, was auf den Einsatz eines DSL-Routers zurückzuführen ist. (Bei einem Vergleichstest von einem anderen Netzwerk aus wurde die MSS auf 1460, also 1500−40, gesetzt.)

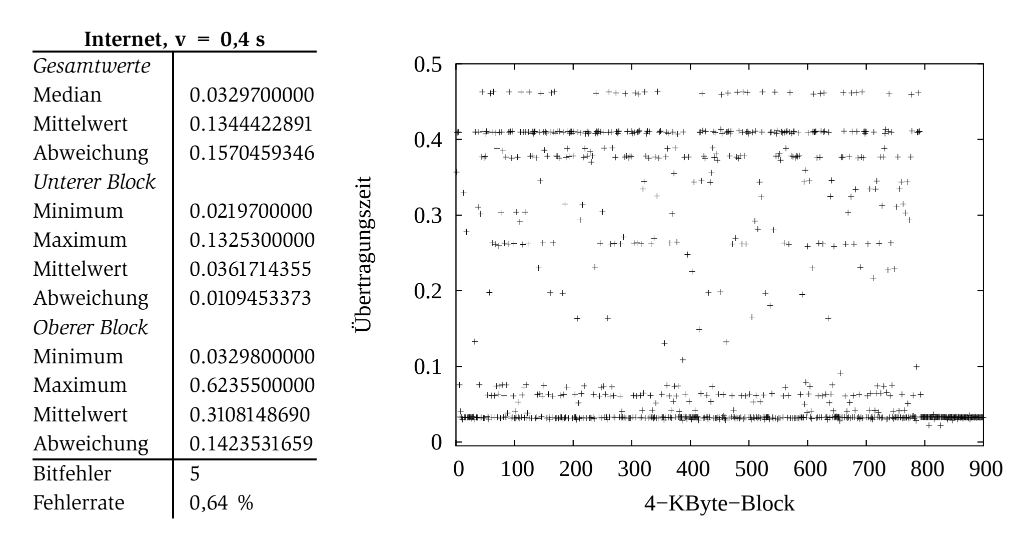
Die vollständig unverzögerte Übertragung der 5 MByte großen Testdatei benötigte über diese Verbindung 42 s; bei Verzögerung von durchschnittlich jedem zweiten 4-KByte-Block um 0,5 s erhöhte sich die Übertragungszeit auf 322 s.

Das liegt im Rahmen der Erwartungen: Die 5 MByte große Datei besteht aus 1280 Blöcken zu je 4 KByte; wird jeder zweite (also 640 Blöcke) um 0,5 s verzögert, ergibt sich eine erwartete Übertragungszeit von 42 s $+\ 640 \times 0,5$ s $= 362$ s.

4.5.2 Messergebnisse

Es werden hier nur einige der Messungen im Detail besprochen; das folgende Unterkapitel 4.5.3 (ab Seite 76) gibt eine Übersicht der Messungen.

0,4 Sekunden: Bei einer Verzögerung von $v = 0,4$ s war die Übertragung nur zu 0,64 % fehlerbehaftet – vergleicht man die Abbildung 4.9 (Seite 75) zu diesem Test mit erfolgreichen Tests im lokalen Netzwerk, fällt auf, dass die dort übliche starke Trennung in einen oberen und einen unteren Block entfällt. Zwar gibt es knapp oberhalb von 0,4 eine Häufung von Messwerten, aber ein Großteil der Werte ist über den gesamten Bereich knapp oberhalb von 0,03 bis 0,6 verteilt.

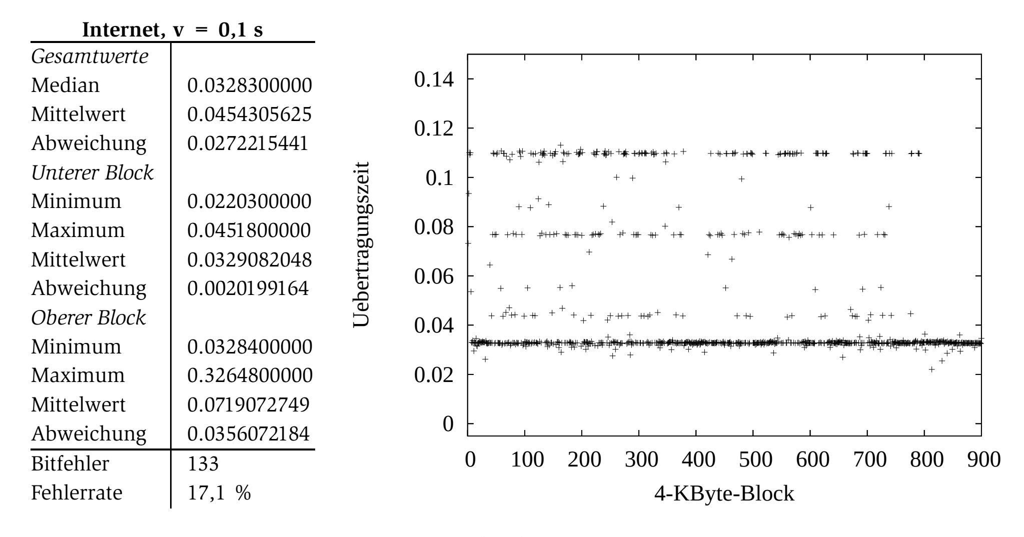


Internet, v = 0,4 s	
Gesamtwerte	
Median	0.0329700000
Mittelwert	0.1344422891
Abweichung	0.1570459346
Unterer Block	
Minimum	0.0219700000
Maximum	0.1325300000
Mittelwert	0.0361714355
Abweichung	0.0109453373
Oberer Block	
Minimum	0.0329800000
Maximum	0.6235500000
Mittelwert	0.3108148690
Abweichung	0.1423531659
Bitfehler	5
Fehlerrate	0,64 %

Abbildung 4.9: Messwerte und Grafik für den Test im Internet mit $v = 0,4$ s.

0,1 Sekunden: Mit $v = 0,1$ s Verzögerung ergaben sich die Werte aus Abbildung 4.10 auf Seite 75. Die Fehlerrate von 17,1 % macht den Kanal nahezu unbrauchbar.

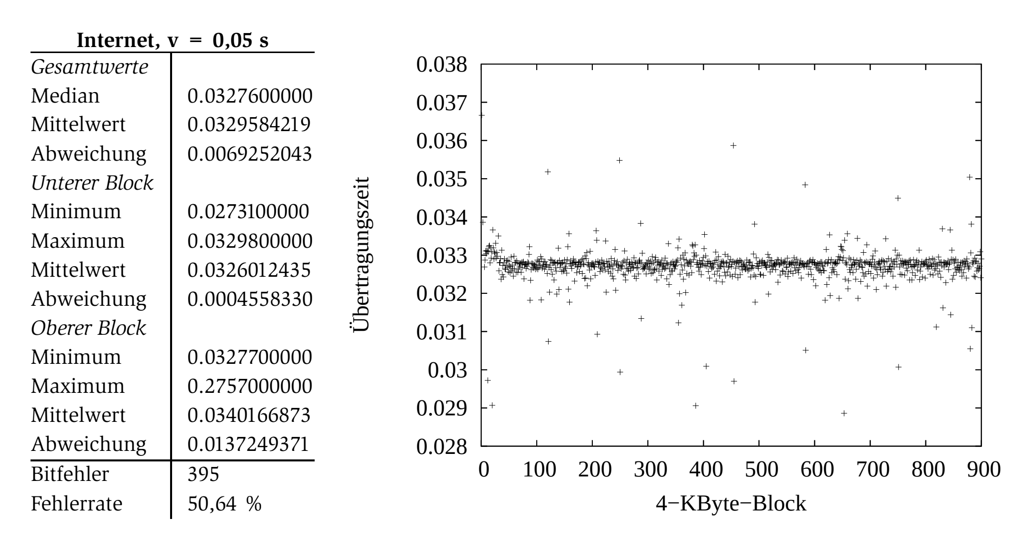
Vergleicht man die Abbildung und die Abweichungen in oberen Block mit den korrespondierenden Werten aus dem Test im lokalen Netz (Beschreibung auf Seite 71; Abbildung 4.6), sieht man auch hier die erheblich größere Streuung der Werte: Während der obere Block im lokalen Netzwerk eine Abweichung von nur 0,001 s aufweist, sind es im Internet-Szenario 0,03 s.



Internet, v = 0,1 s	
Gesamtwerte	
Median	0.0328300000
Mittelwert	0.0454305625
Abweichung	0.0272215441
Unterer Block	
Minimum	0.0220300000
Maximum	0.0451800000
Mittelwert	0.0329082048
Abweichung	0.0020199164
Oberer Block	
Minimum	0.0328400000
Maximum	0.3264800000
Mittelwert	0.0719072749
Abweichung	0.0356072184
Bitfehler	133
Fehlerrate	17,1 %

Abbildung 4.10: Messwerte und Grafik für den Test im Internet mit $v = 0,1$ s.

0,05 Sekunden: Eine Verzögerung von $v = 0,05$ s führte zur völligen Unbrauchbarkeit des Kanals: Mit 50,64 % Fehlerrate war der Kanal vollständig gestört. Abbildung 4.11 auf Seite 76 zeigt, dass sich fast alle Messwerte in einem kleinen Intervall um 0,0033 befinden – die erwartete Aufteilung in zwei Blöcke gab es nicht. (Die Abbildung zeigt auf der y-Achse nur den relevanten Ausschnitt von 0,028 s bis 0,038 s, da unter- und oberhalb kaum Messwerte lagen.)



Internet, v = 0,05 s	
Gesamtwerte	
Median	0.0327600000
Mittelwert	0.0329584219
Abweichung	0.0069252043
Unterer Block	
Minimum	0.0273100000
Maximum	0.0329800000
Mittelwert	0.0326012435
Abweichung	0.0004558330
Oberer Block	
Minimum	0.0327700000
Maximum	0.2757000000
Mittelwert	0.0340166873
Abweichung	0.0137249371
Bitfehler	395
Fehlerrate	50,64 %

Abbildung 4.11: Messwerte und Grafik für den Test im Internet mit $v = 0,05$ s.

4.5.3 Diskussion der Ergebnisse

Abbildung 4.12 (auf der folgenden Seite) zeigt die Fehlerquoten für die verschiedenen Verzögerungszeiten – im oberen Graphen mit linearer y-Achse und im unteren Graphen mit logarithmischer y-Achse. Zum Vergleich wurde in beide Graphen die Exponentialfunktion $f(x) = \exp(18 * (0,26 - x))$ eingezeichnet.

An der Grafik kann man erkennen, dass die Fehlerquote exponentiell steigt, wenn die Verzögerung reduziert wird.

Für $v = 0,05$ s (und kleinere v) lag die Fehlerrate bei 50 % – hier war keine Unterscheidung zwischen verzögerten und unverzögerten Blöcken mehr möglich, und der Proxy auf Empfängerseite interpretierte alle Blöcke gleich. Da die Nullen und Einsen in der verdeckten Nachricht annähernd gleich verteilt sind, ist 50 % das schlechtestmögliche Ergebnis.

Bei den Tests mit 0,06 s $\leq v \leq$0,15 s wurden 39,18 % – 7,36 % der Bits fehlerhaft übertragen. Ab $v = 0,2$ s lag die Fehlerquote unter 3,54 %, so dass die Hoffnung bestand, dass der Einsatz eines Fehlerkorrekturverfahrens für meist korrekte Übertragungen sorgen könnte. (Mehr dazu im folgenden Unterkapitel.) Selbst bei einer halben Sekunde Verzögerung traten noch Fehler auf – im Mittel waren 0,1 % der Bits falsch.

Warum kann keine Fehlerrate erreicht werden, die deutlich über 50 % liegt? Die Analyse der Übertragungszeiten basiert darauf, die Messwerte in zwei Bereiche unterteilen zu können, von denen Werte im oberen Teil als verzögert und Werte im unteren Teil als unverzögert interpretiert werden. Das schlechtestmögliche Ergebnis würde bei einer Verzögerung von 0 s erreicht: Dann (aber auch schon bei ausreichend kleinen Werten $\neq 0$ s) gibt es diese Trennung nicht mehr, und alle empfangenen Werte werden als unverzögert (oder alle als verzögert) interpretiert. Da die Quelle annähernd gleich verteilt Nullen und Einsen überträgt, wird die Hälfte der Werte gekippt; das ergibt 50 % Fehlerrate.

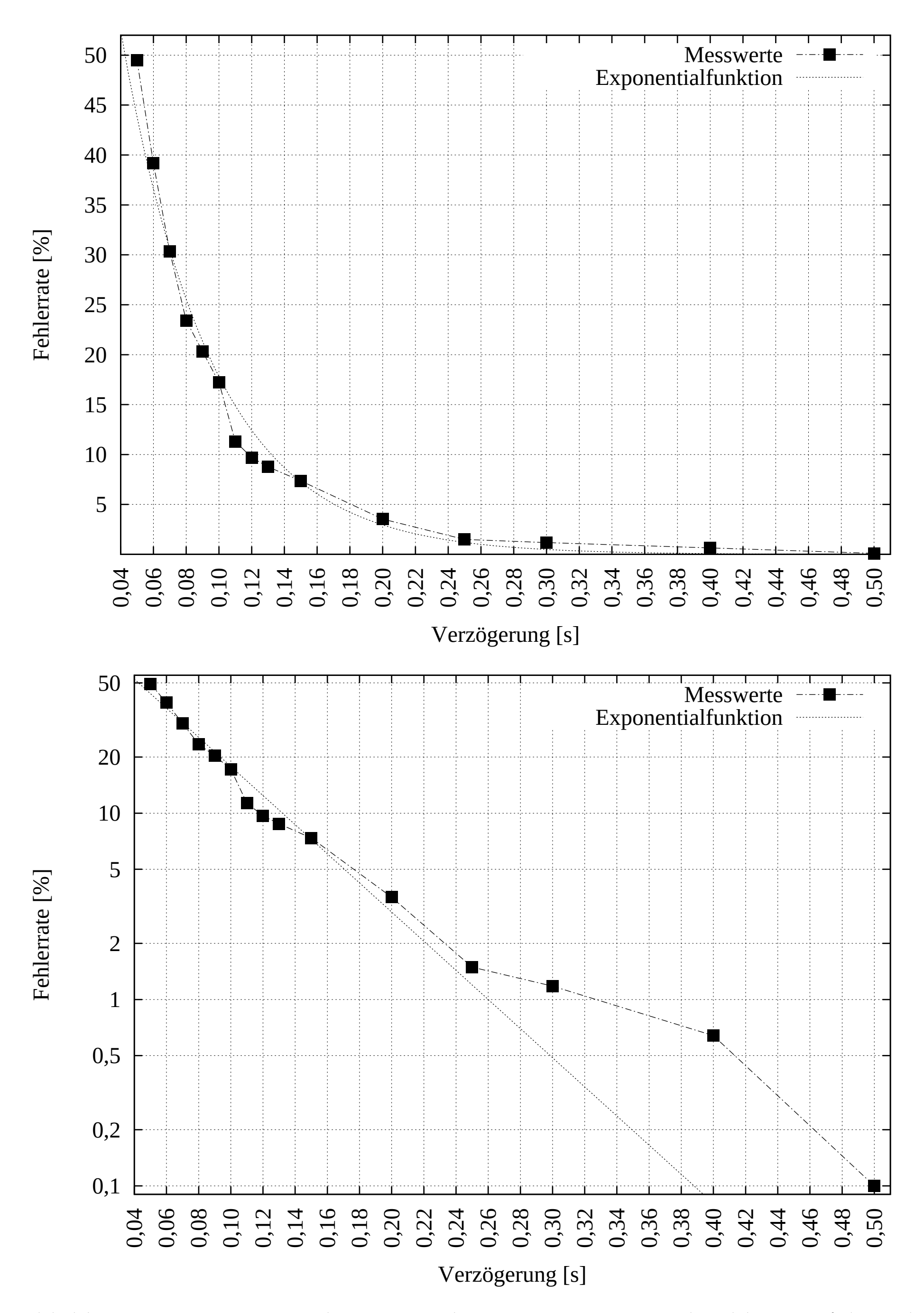


Abbildung 4.12: Zusammenhang zwischen Verzögerung und Fehlerrate (oben: lineare y-Achse, unten: logarithmische y-Achse; Verbindungslinien nur zur besseren Veranschaulichung).

Eine Fehlerwahrscheinlichkeit von 100 % würde einem invertierenden Kanal entsprechen, der alle Bits kippt – 100 % Fehlerrate sind also genauso schwer zu erreichen wie 0 % Fehlerrate.

4.5.4 Tests mit Fehlerkorrektur

Im Anschluss an die Testreihe wurde das in Kapitel 2.7 (ab Seite 26) beschriebene Fehlerkorrekturverfahren in die Kommunikation integriert. Eine Wiederholung der Messungen mit aktivierter Fehlerkorrektur führte nicht zu den erwarteten deutlichen Verbesserungen.

Abbildung 4.13 auf Seite 78 zeigt den interessanten Bereich mit Verzögerungen zwischen 0,1 s und 0,5 s: Für 0,4 s und 0,5 s war die Übertragung mit aktivierter Fehlerkorrektur fehlerfrei, während ohne Fehlerkorrektur Übertragungsfehler auftraten. Für Verzögerungen von 0,15 s, 0,2 s und 0,3 s gab es trotz Fehlerkorrektur Übertragungsfehler, die Quote der korrekt übertragenen Daten war aber besser als ohne Fehlerkorrektur.

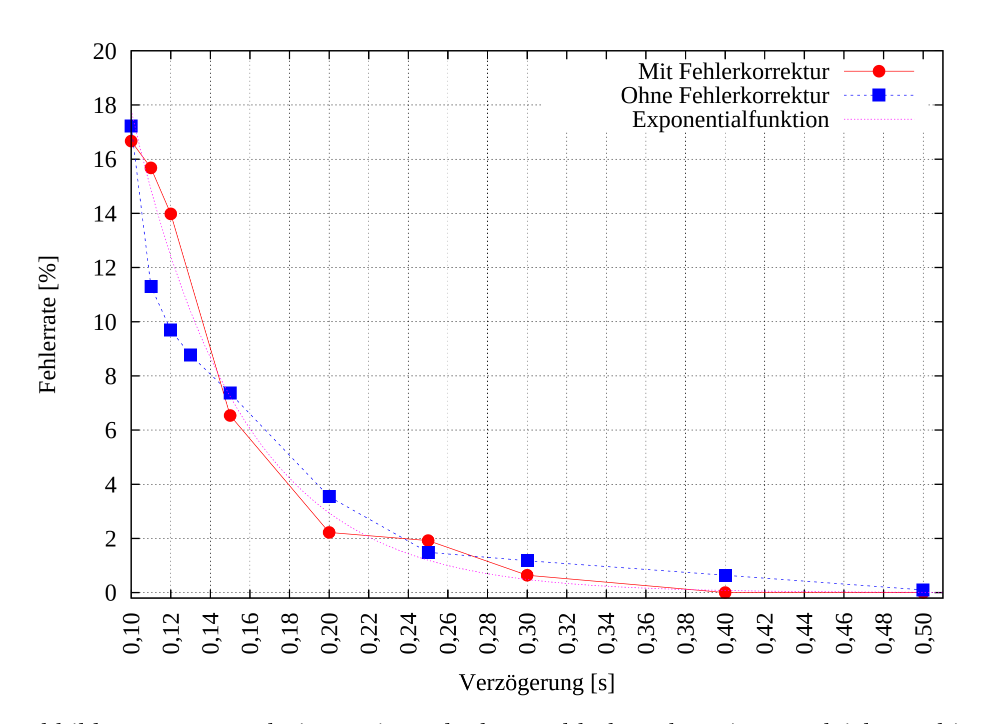


Abbildung 4.13: Ergebnisse mit und ohne Fehlerkorrektur im Vergleich; Verbindungslinien nur zur besseren Veranschaulichung.

Bei 0,11 s und 0,12 s Verzögerung waren die Ergebnisse mit Fehlerkorrektur sogar schlechter als ohne – hier ist allerdings zu beachten, dass die Datenbasis mit jeweils fünf Messungen (im Fall ohne Fehlerkorrektur) bzw. vier Messungen (im Fall mit Fehlerkorrektur) recht geringe Aussagekraft hat.

Tabelle 4.2 auf der folgenden Seite führt neben den Fehlerraten auch die Übertragungszeiten (für die Übertragungen ohne Fehlerkorrektur) auf.

Aufgrund der geringen oder fehlenden Verbesserungen wurde das Fehlerkorrekturverfahren einer statistischen Analyse unterzogen. Die Fragestellung war, mit welcher Wahrscheinlichkeit ein (6-Bit-) Zeichen vollständig korrekt übertragen wird.

Dazu sei p die Fehlerwahrscheinlichkeit für die Übertragung eines Bits.

Verzögerung	Fehler (mit Korr.)	Fehler (ohne Korr.)	Zeit (min)
0,05	48,33	49,49	0:45
0,06	40,38	39,18	0:46
0,07	32,91	30,35	0:55
0,08	24,40	23,41	1:00
0,09	20,04	20,33	1:04
0,1	16,67	17,23	1:10
0,11	15,68	11,31	1:16
0,12	13,98	9,69	1:31
0,15	6,54	7,36	1:41
0,2	2,22	3,54	2:12
0,25	1,92	1,49	2:44
0,3	0,64	1,18	3:15
0,4	0,00	0,64	4:19
0,5	0,00	0,10	5:22

Tabelle 4.2: Übersicht der Ergebnisse mit und ohne Fehlerkorrektur.

- Im Fall ohne Fehlerkorrektur werden sechs Bits übertragen. Die Wahrscheinlichkeit, dass alle sechs Bits korrekt den Empfänger erreichen, ist einfach $p_1 := (1-p)^6$.
- Im Fall mit Fehlerkorrektur werden die sechs Bits um vier zusätzliche Bits ergänzt. Die Fehlerkorrektur kann einen Fehler korrigieren, ab zwei Fehlern ist das Ergebnis falsch. Damit ist die Wahrscheinlichkeit für die korrekte Übertragung die Summe zweier Teilwahrscheinlichkeiten:

  – Übertragung aller zehn Bits ohne Fehler
  – Übertragung der zehn Bits mit einem Fehler

  Die erste Teilwahrscheinlichkeit ist analog zum oberen Fall $(1-p)^{10}$; die zweite Teilwahrscheinlichkeit ist $\sum_{i=1}^{10} p(1-p)^9$. (Jeder der zehn Summanden in diesem Ausdruck steht für die Wahrscheinlichkeit, dass Bit Nummer i falsch übertragen und alle anderen Bits korrekt übertragen werden.)

  Da die beiden Teilwahrscheinlichkeiten disjunkte Ereignismengen beschreiben, kann man sie summieren; es ergibt sich

$$p_2 := (1-p)^{10} + \sum_{i=1}^{10} p(1-p)^9 = (1-p)^{10} + 10p(1-p)^9$$

Für die Fehlerraten p aus den Tests ohne Fehlerkorrektur wurden die Werte p_1 und p_2 berechnet; Tabelle 4.3 (auf der folgenden Seite) listet die Fehlerraten $(1-p_i)$ auf.

Mit steigender Bitfehlerquote sinkt also der Nutzen der Fehlerkorrektur: Sie kann nur bei sehr kleinen Fehlerraten deutliche Verbesserungen bewirken. Das entspricht der Erwartung, weil das Verfahren so konstruiert ist, dass es maximal einen Fehler in zehn übertragenen Bits verträgt.

Verz.	p	$1-p_1$	$1-p_2$	$(1-p_2)/(1-p_1)$
–	0,01 %	0,060 %	0,000 %	0,001
–	0,05 %	0,300 %	0,001 %	0,004
0,50 s	0,10 %	0,599 %	0,004 %	0,007
0,40 s	0,64 %	3,779 %	0,178 %	0,047
0,30 s	1,18 %	6,874 %	0,588 %	0,086
0,25 s	1,49 %	8,614 %	0,923 %	0,107
0,20 s	3,54 %	19,447 %	4,668 %	0,240
0,15 s	7,36 %	36,789 %	16,455 %	0,447
0,12 s	9,69 %	45,748 %	25,191 %	0,551
0,11 s	11,31 %	51,332 %	31,487 %	0,613
0,10 s	17,23 %	67,846 %	53,493 %	0,788
0,09 s	20,33 %	74,428 %	63,407 %	0,852
0,08 s	23,41 %	79,815 %	71,824 %	0,900
0,07 s	30,35 %	88,584 %	85,606 %	0,966
0,06 s	39,18 %	94,939 %	94,846 %	0,999
0,05 s	49,49 %	98,339 %	98,833 %	1,005

Tabelle 4.3: Fehlerraten mit und ohne Fehlerkorrektur.

Akzeptiert man, dass 10 % der (ASCII-) Zeichen falsch übertragen werden, muss man nach den gemessenen und berechneten Werten eine Verzögerung von knapp unter 0,25 s (ohne Fehlerkorrektur) oder von knapp unter 0,2 s (mit Fehlerkorrektur) wählen.

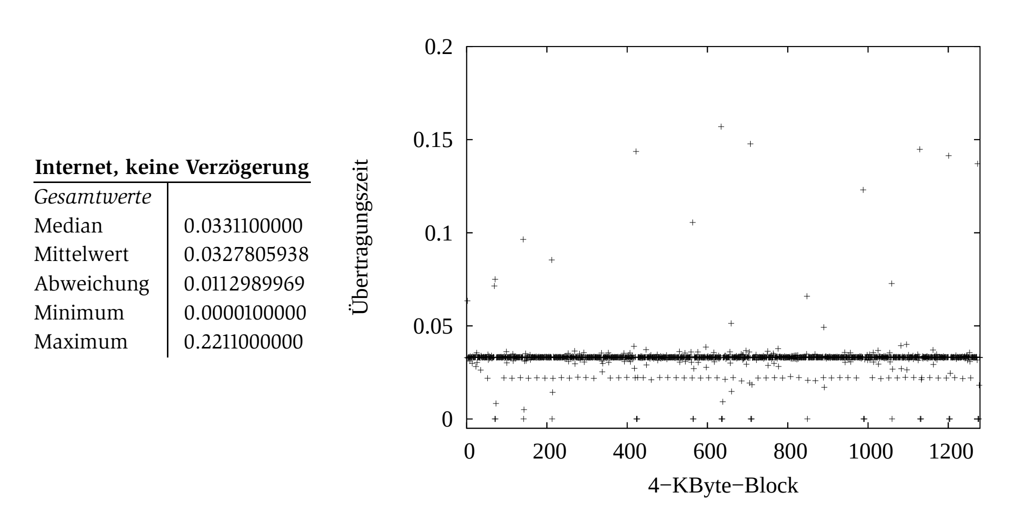
4.5.5 Eigenverursachte Fehler

Interessant war auch die Fragestellung, wie stark die zeitlichen Schwankungen sind, die ohne verdeckte Kommunikation beim Herunterladen der Testdateien auftreten.

Abbildung 4.14 (auf der folgenden Seite) zeigt das Diagramm für eine Übertragung, bei der der Empfänger nicht als Kommunikationspartner für die verdeckte Kommunikation eingetragen war. Die meisten Zeiten liegen in der Nähe von 0,03, einige „Ausreißer“ erreichten Werte von bis zu ca. 0,16. Interpretiert man Werte ab 0,05 als zu lange Übertragungszeiten, gab es in fünf Testdurchläufen 15, 14, 17, 15 und 16, im Schnitt 15,4 falsche Bits. Bezogen auf 1280 übertragene Blöcke entspricht das einer durchschnittlichen Fehlerquote von 1,2 %.

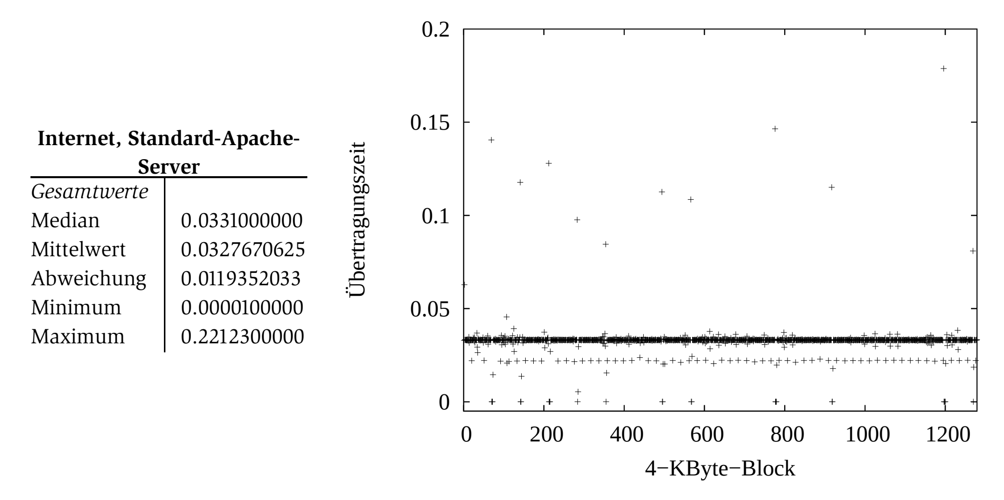
Das bedeutet aber nicht, dass dadurch eine untere Schranke für die Fehlerrate angegeben wäre, sondern bestätigt lediglich das Ergebnis, dass die Verzögerung ausreichend hoch gewählt werden muss: Die Tests mit größeren Verzögerungen ergaben niedrige Fehlerraten.

Abbildung 4.15 (auf der folgenden Seite) zeigt die gleichen Daten für einen ungepatchten Apache-Server – also für einen Server, der keine Anfragen an der Daemon-Prozess richtet.



Internet, keine Verzögerung	
Gesamtwerte	
Median	0.0331100000
Mittelwert	0.0327805938
Abweichung	0.0112989969
Minimum	0.0000100000
Maximum	0.2211000000

Abbildung 4.14: Messwerte und Grafik für den Test im Internet ohne Verzögerung.



Internet, Standard-Apache-Server	
Gesamtwerte	
Median	0.0331000000
Mittelwert	0.0327670625
Abweichung	0.0119352033
Minimum	0.0000100000
Maximum	0.2212300000

Abbildung 4.15: Messwerte und Grafik für den Test im Internet mit einem Standard-Apache-Server.

Mit beiden Apache-Servern (gepatcht und ungepatcht) benötigte die Übertragung der 5-MByte-Datei ca. 42 Sekunden, die Anfragen beim Daemon-Prozess verursachen also keine messbare Verzögerung.

Die Anzahl der Blöcke mit Transferzeiten von über 0,05 s war in fünf durchgeführten Tests 13, 11, 16, 11 und 16, im Schnitt 13,4; das entspricht einer Fehlerrate von 1,05 %, die geringfügig unterhalb der Rate von 1,2 % beim gepatchten Web-Server liegt. Diese Abweichung ist allerdings gering und kann der geringen Stichprobenzahl zugeschrieben werden.

4.6 Test im Internet unter Last

Um eine Belastungssituation für den Server zu simulieren, wurden fünf kleinere (512 KByte große) Testdateien erzeugt, die simultan von zwei Rechnern herunter geladen wurden, während der Test ausgeführt wurde.

Diese Testdateien wurden mit

```
dd if=/dev/urandom of=random.data.1 bs=1024 count=512
for i in 2 3 4 5; do cp random.data.1 random.data.$i; done
```

erzeugt. Das Skript `stresstest-single.sh` zieht zehnmal nacheinander die Testdatei mit der als Argument angegebenen Nummer und löscht sie danach wieder.

```
#!/bin/bash
# stresstest-single.sh
NUM=$1   # random.data.$NUM ziehen
FNAME=random.data.$NUM
for i in 1 2 3 4 5 6 7 8 9 10; do
  wget http://hgesser.com:8080/$FNAME &> /dev/null
  rm $FNAME
done
```

Das Steuerprogramm `stresstest.sh` schließlich startet zeitversetzt fünf Instanzen von `stresstest-single.sh`.

```
#!/bin/bash
# stresstest.sh

cd /tmp
mkdir -p cctest
cd cctest
~/bin/stresstest-single.sh 1 &
sleep 3
~/bin/stresstest-single.sh 2 &
sleep 3
~/bin/stresstest-single.sh 3 &
sleep 3
~/bin/stresstest-single.sh 4 &
sleep 3
~/bin/stresstest-single.sh 5 &
echo Alle Prozesse laufen:
ps auxww|grep stress
```

Die erhöhte Last hatte keine Auswirkung auf die Messergebnisse, auf eine Darstellung der (bis auf statistische Abweichungen gleichen) Daten wird daher hier verzichtet.

Interessant ist die Frage, ob es unter der Last einer sehr stark besuchten Web-Seite mit einer Millionen Seitenzugriffen pro Tag (das entspricht ca. 11,57 Zugriffen pro Sekunde) zu anderen Ergebnissen kommt – laut von Google veröffentlichten Statistiken [Goo] hatte etwa diese Suchmaschine bereits im Februar 2001 über 100 Millionen Zugriffe am Tag (die sich allerdings auf mehrere Web-Server verteilen). Mangels Zugriff auf einen entsprechend stark frequentierten Server konnte diese Frage aber nicht beantwortet werden.

4.7 Beobachtbarkeit

Das Wesen eines verdeckten Kanals ist die unauffällige Übertragung von Daten – daher ist es wichtig, die Beobachtbarkeit zu prüfen.

Für eine Verzögerungszeit von 0,3 s, die eine geringe Fehlerquote erlaubt, wurde der Netzwerkverkehr mit dem Hilfsprogramm `ethereal` [eth] untersucht; dem wurden Vergleichsmessungen ohne Verzögerung gegenübergestellt.

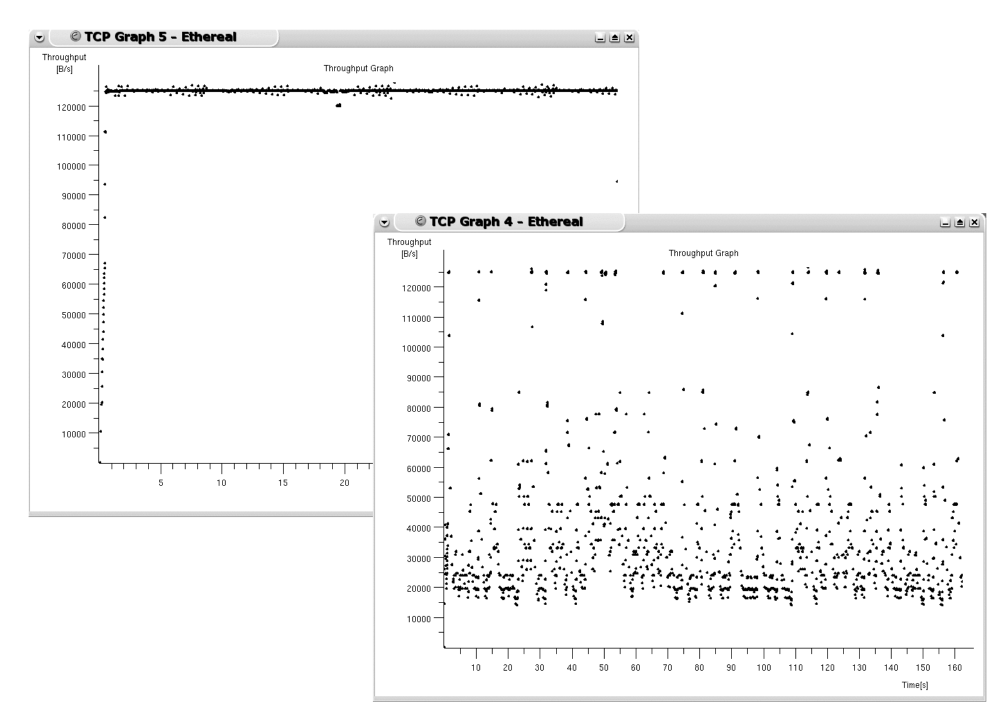


Abbildung 4.16: Von `ethereal` beobachteter Datendurchsatz – links: unverzögert, rechts: verzögert.

Wird ein verdeckter Zeitkanal vermutet und mit Ethereal danach gesucht, fallen Unterschiede beim Datendurchsatz auf. Abbildung 4.16 zeigt links die Datendurchsatzstatistik (*Throughput Graph*) bei einer unverzögerten Übertragung. Deutlich erkennbar liegt die Rate während des gesamten Transfers einigermaßen konstant um 125000 Byte/s. Die rechte Abbildung zeigt den von Ethereal ermittelten Datendurchsatz bei der Übertragung mit verdecktem Kanal – hier ergibt sich ein anderes Bild, der Maximalwert von 125000 Byte/s wird nur in wenigen Fällen angenommen, die Werte sind zu einem großen Teil über den Bereich von 15000 bis 85000 Byte/s verschmiert.

Aus der Beobachtung ergibt sich aber zunächst nur, dass der Web-Server mit stark schwankender Datenrate arbeitet – dies könnte auch hoher Server-Auslastung zugeschrieben werden.

Ethereal erlaubt bei der Anzeige der Pakete auch die Darstellung der seit dem letzten betrachteten Paket vergangenen Zeit. Auch hier kann man verzögerte Pakete beobachten; Abbildung 4.17 zeigt ein Beispielintervall. Während im unverzögerten

Fall die Zeitabstände meist um 0,01 s liegen, gibt es bei der verzögerten Übertragung viele Pakete, bei denen der Abstand zum vorherigen um 0,25 s und damit in der Größenordnung der Verzögerung von 0,3 s liegt.

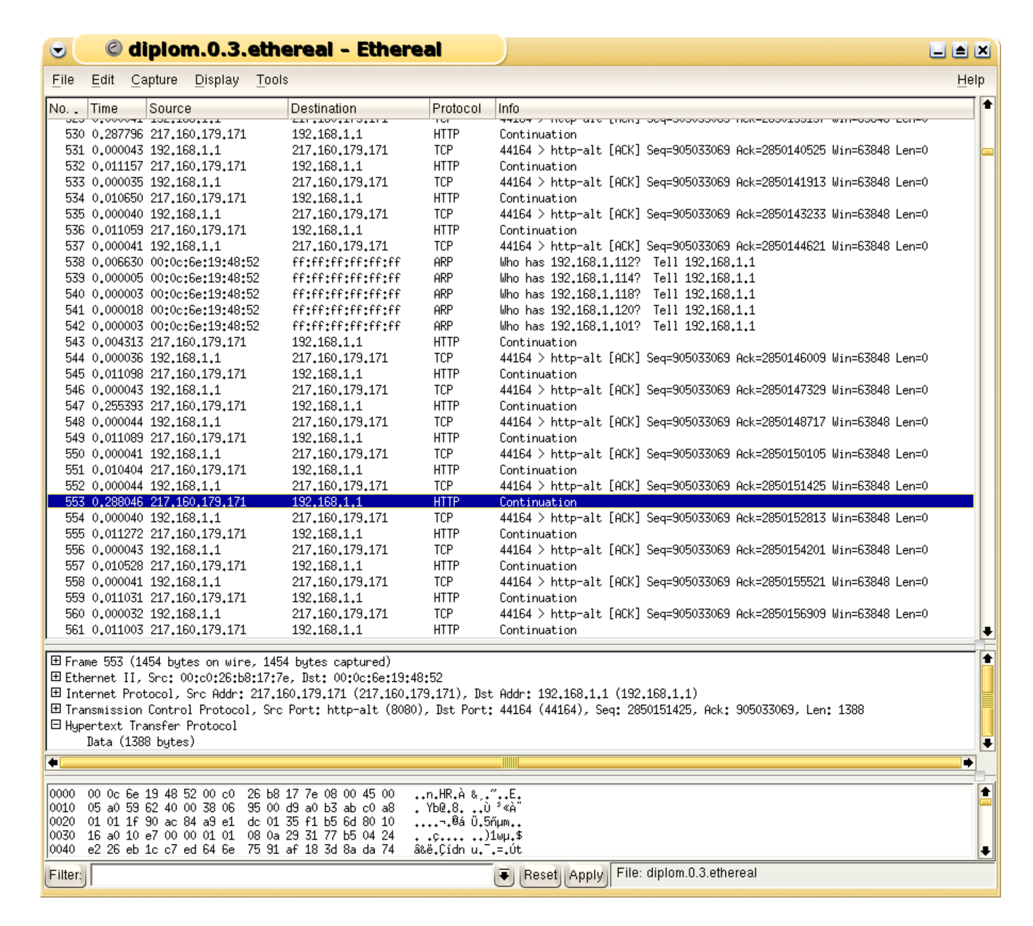


Abbildung 4.17: Netzwerkverkehr bei Übertragung mit Verzögerung in der Übersicht von `ethereal`.

4.8 Kapazitätsbetrachtungen

In Kapitel 2.5.4 ab Seite 23 wurde ein theoretisches Resultat für die Kapazität des implementierten Kanals vorgestellt:

Bei einer unverzögerten Download-Zeit von 0,0328 s pro 4-KByte-Block und einer Verzögerung um den neunfachen Wert ergibt sich als Kapazität $C_{1,10} = 0,26$, d. h. es werden maximal 0,26 Bits pro Zeiteinheit (von 0,0328 s) über diesen Kanal übertragen.

Die theoretischen Ergebnisse werden in diesem Abschnitt mit den Messwerten verglichen.

Bei 0,3 s Verzögerung benötigte die gesamte Übertragung 3:15 Minuten, also 195 s. In dieser Zeit wurden 1280 4-KByte-Blöcke übertragen. $195/0.0328 \approx 5945$, d. h., die

Übertragung hat 5945 Zeiteinheiten benötigt. Multipliziert man diese Zahl mit der berechneten Kapazität 0,26, ergibt sich der Wert ≈ 1546: Damit liegt die Anzahl von 1280 Datenbits (auf dem verdeckten Kanal) unterhalb der theoretischen Kapazität. Die Fehlerrate lag für 0,3 s (ohne Fehlerkorrektur) bei 1,18 % und damit im Bereich der eigenverursachten Fehler (siehe Kapitel 4.5.5).

Für eine Verzögerung von 0,1 s kann man die gleiche Berechnung durchführen: $0,1 \approx 0,984 = 0.0328 * 3$. Nach Kapitel 2.5.4 berechnet sich die Kapazität als Logarithmus der positiven Nullstelle von $x^4 - x^3 - 1$. Eine Berechnung in Octave [Oct, Eat97] fand ≈ 1.38028 als Nullstelle, die Kapazität ist damit ungefähr $\log 1.38028 = 0,46 =: C_{1,4}$.

Bei 0,1 s benötigte die gesamte Übertragung 1:10 Minuten, das sind 70 s. Mit den gleichen Berechnungen wie oben ergeben sich $70/0.0328 \approx 2134$ und $2134 * 0,46 \approx 982$. Die Kapazitätsuntersuchung zeigt also, dass über diesen Kanal 982 Bit übertragen werden können – es wurde aber versucht, 1280 zu senden: Damit ist der Kanal um $1280/982 - 1 \approx$ 30,3 % überlastet, was die hohe Fehlerrate (17,23 %) erklärt.

Für 0,06 s Verzögerung ergibt sich ein ähnliches Bild: 0,06 ist (sehr grob) annähernd $0,0328 * 2$. Damit muss die positive Nullstelle von $x^3 - x^2 - 1$ gesucht werden, die (wieder mit Octave berechnet) ≈ 1.46557 ist. Es ist $\log 1.46557 \approx 0,55 =: C_{1,3}$. Die Übertragungszeit für diese Verzögerung war 46 s, damit berechnet man die Kapazität über $46/0.0328 \approx 1402$ und $1402 * 0,55 \approx 771$. Der Kanal ist also um $1280/771 - 1 \approx 66$ % überlastet, die Fehlerrate lag bei 39,2 %.

Für einen grafischen Vergleich von (halber) Kanalüberlastung und Fehlerrate wurden noch die Messwerte bei einer Verzögerung von 0,2 s berücksichtigt – die anderen Werte sind zunächst für diese Untersuchung ungeeignet, da die Verzögerungszeiten keine annähernden Vielfachen von 0,0328 sind und die theoretischen Aussagen nur für ganzzahlige Vielfache gelten.

Es ist $0,2 \approx 6 * 0,0328$, die positive Nullstelle von $x^7 - x^6 - 1$ ist $\approx 1,25542$, der Logarithmus ist $\approx 0,328$. Die Übertragungszeit war 2:12 Minuten, also 132 s. $132/0.0328 \approx 4024$ und $4024 * 0,328 \approx 1320$. Der Kanal ist damit nicht überlastet $(1320 > 1280)$.

Um noch einen weiteren Wert zu gewinnen, wurde für 0,12 s $(0,12 \approx 3\frac{2}{3} * 0,0328)$ die Zeiteinheit auf $0,0328/3 = 0,0109\overline{3}$ geändert. Damit ist in der Notation aus Kapitel 2.5.4 $t_0 = 3$ und $t_1 \approx 11 + 3 = 14$. Gesucht wird jetzt die Nullstelle von $1 - x^{-3} - x^{-14}$; Multiplikation mit x^{14} ergibt $0 = x^{14} - x^{11} - 1$. Octave berechnet als Nullstelle 1,10275, der Logarithmus ist 0,1411 – das ist die Kapazität in Bit pro Zeiteinheit. Zur Normierung muss dieser Wert wieder mit 3 multipliziert werden (weil für diese Berechnungen gedrittelte Zeiteinheiten verwendet wurden).

Die Übertragungszeit war 1:31 Minuten, also 91 s. $91/0,0328 \approx 2774$ und $2774 * 0,1411 * 3 \approx 1174$. Der Kanal ist damit um $1280/1174 - 1 \approx 9,03$ % überlastet.

Die grafische Auswertung zeigt Abbildung 4.18 (auf der folgenden Seite): Auf den beiden Kurven liegen die gemessenen Fehlerraten und die (halbierten) berechneten Kanalüberlastungen für verschiedene Verzögerungszeiten v.

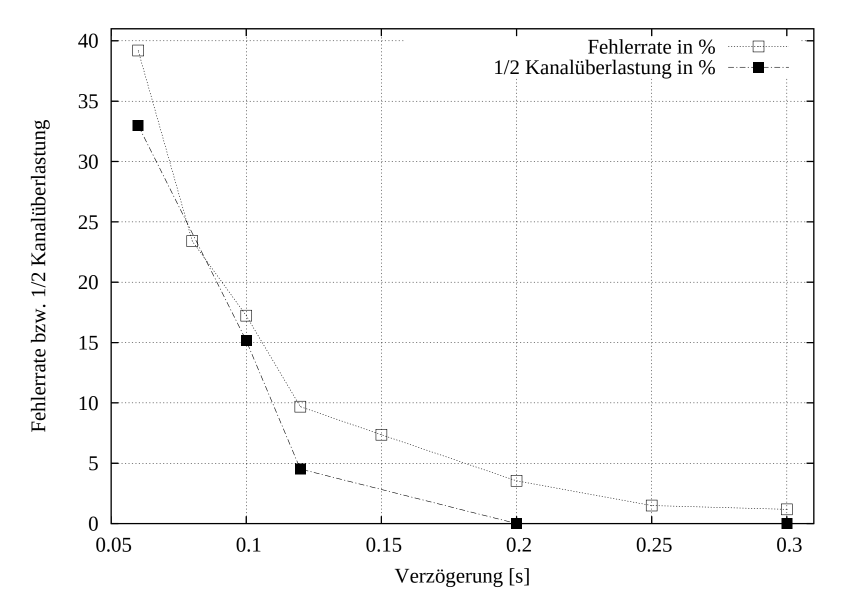


Abbildung 4.18: Fehlerrate und (halbierte) Kanalüberlastung im Vergleich; Verbindungslinien nur zur Veranschaulichung.

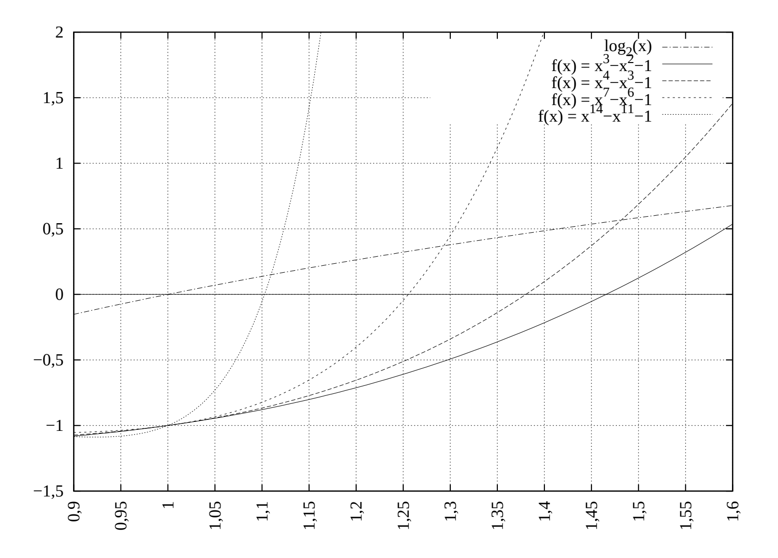


Abbildung 4.19: Nullstellensuche in $x^3 - x^2 - 1$, $x^4 - x^3 - 1$, $x^7 - x^6 - 1$ und $x^{14} - x^{11} - 1$.

Abbildung 4.19 zeigt die Graphen der verwendeten Polynome in der Nähe ihrer positiven Nullstellen. Zu beachten ist auch, dass die Fehlerrate nahe an der Hälfte der prozentualen Überlastung liegt: Das entspricht der maximalen Fehlerrate von 50 % bei totaler Störung des Kanals. Ist der Kanal gestört, kann nicht mehr zwischen 0 und 1 unterschieden werden, so dass bei annähernder Gleichverteilung der gesendeten Bits immer die Hälfte der empfangenen Bits korrekt ist.

4.8.1 Kapazität im lokalen Netzwerk

Im lokalen Netzwerk kann man die Ergebnisse ebenfalls prüfen: Die Verzögerung von 0,03 s entspricht ungefähr einer Ver-55-fachung der Übertragungszeit, also muss hier $C_{1,55}$ berechnet werden. Das quadratische Polynom $x^{55} - x^{54} - 1$ hat die positive Nullstelle ≈ 1.05513, der Logarithmus ist $\log 1.05513 \approx 0,77$.

Der Berechnungsfaktor 0,0328 ist hier durch die kleinere Übertragungszeit pro Paket im lokalen Netzwerk zu ersetzen; aus der vollständig unverzögerten Übertragung wurde er als $\approx$ 0,00055 ermittelt.

Die Übertragung benötigte 12,83 s, $12.83/0.00055 \approx 23327$, und $23327 * 0,77 \approx 17962$ – damit ist Platz für mehr als das Zehnfache der übertragenen 1280 Bits.

Auch dieses Ergebnis bestätigt die kaum fehlerbehaftete Übertragung bei 0,03 s Verzögerung im lokalen Netzwerk.

4.8.2 Eine offene Frage

In Kapitel 4.5.3 wurde darauf hingewiesen, dass die Fehlerrate mit größer werdender Verzögerung v exponentiell sinkt; Abbildung 4.12 auf Seite 77 zeigt diesen Zusammenhang anschaulich.

Wünschenswert wäre aber eine Untermauerung dieser Aussage durch einen mathematischen Nachweis.

Ein formaler Beweis wurde nicht gefunden, darum präsentiert dieses Kapitel einige zusätzliche Überlegungen, die dabei helfen, das Problem „einzukreisen“.

Bei den ursprünglichen Tests wurde eine 5 MByte große Datei vom Apache-Server heruntergeladen. Diese wurde in 1280 Blöcke (zu je 4 KByte) zerlegt. Die zufällig generierte Testnachricht hatte nach der Kodierung in Bits die Länge 792, davon 412 Nullen (52 %) und 380 Einsen (48 %).

Da mit 792 zu übertragenden Bits nicht die volle Blockzahl von 1280 ausgeschöpft wurde, sendete der Server die restlichen $1280 - 792 = 488$ Blöcke unverzögert. In der Summe gab es damit 380 verzögerte (29,69 %) und 900 unverzögerte Pakete (70,31 %). Um diese ungleiche Verteilung von Nullen und Einsen auszugleichen, wurden die Tests für alle Verzögerungszeiten mit einer 792 Blöcke (ca. 3,1 MByte) großen Datei wiederholt – wie vorher wurden auch diesmal je fünf Messungen mit gleicher Verzögerung durchgeführt.

Die gemittelten Messwerte für die Fehlerrate und die Übertragungsdauer zeigt Tabelle 4.4 auf der folgenden Seite. Für die Analyse wurden hier nur die Übertragungszeiten betrachtet. Sie können durch die lineare Funktion $v \mapsto 380v + 4,56$ approximiert werden. Abbildung 4.20 auf der folgenden Seite zeigt die Übertragungszeiten für den ersten Durchlauf (mit 1280 Blöcken) und den neuen Durchlauf (mit 792 Blöcken) sowie den Verlauf zweier linearer Abbildungen, die diese Zeiten approximieren.

Um die gemessenen Fehlerraten mit der halbierten theoretisch hergeleiteten Überlastung des Kanals zu vergleichen, wurden für verschiedene Vielfache der regulären

Verzögerung	Fehler	Zeit (s)
0	(—)	26,33
0,05	49,85	26,53
0,06	38,77	28,36
0,07	30,56	31,35
0,08	24,38	35,03
0,09	20,64	38,92
0,1	18,28	42,63
0,11	16,77	46,34
0,12	11,51	50,20
0,15	8,28	62,00
0,2	4,77	80,50
0,25	3,28	99,60
0,3	2,21	118,57
0,4	1,82	156,57
0,5	1,10	194,71

Tabelle 4.4: Übersicht der Ergebnisse beim Download von 792 Blöcken.

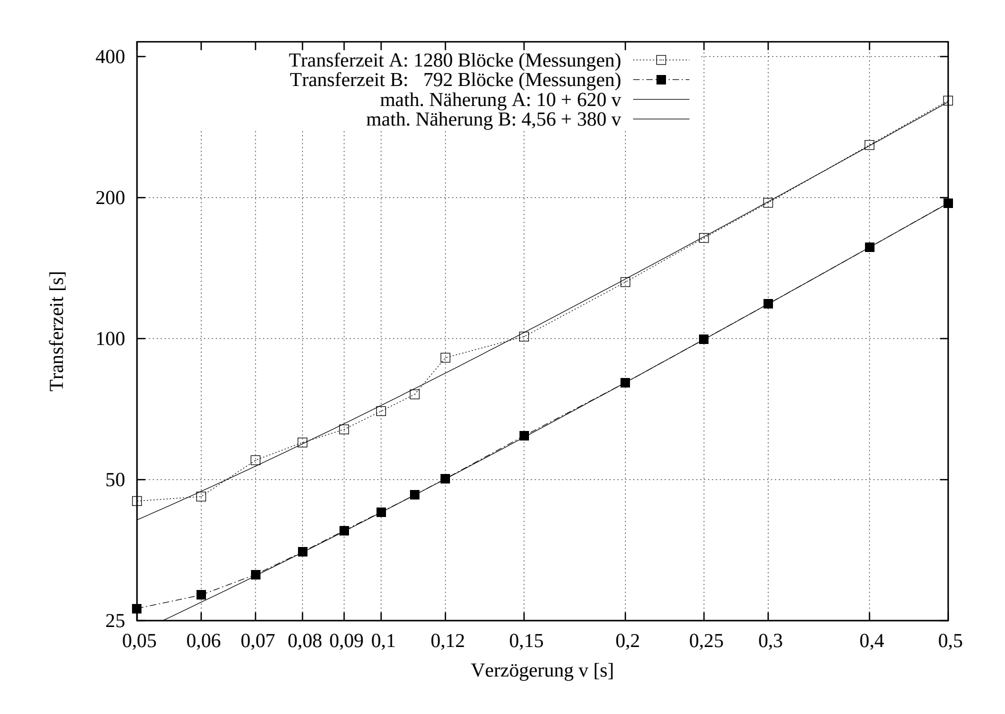


Abbildung 4.20: Transferzeiten approximiert durch lineare Funktionen.

Übertragungszeit $t_0 = 0{,}0332$ s eines einzelnen Pakets die gleichen Berechnungen durchgeführt, wie sie weiter oben beschrieben wurden:

- Die reguläre Übertragungszeit (ohne Verzögerung) ist $t_0 \approx 0{,}0332$.
- Sei $v = (n-1) \times t_0$ die künstliche Verzögerung.

n	$C_n = \log_2 a_n$
3	0,5514618
4	0,4649609
5	0,4056874
6	0,3619928
7	0,3281700
8	0,3010608
9	0,2787579
10	0,2600136
11	0,2440102
12	0,2301415
13	0,2179999
14	0,2072556
15	0,1976767
16	0,1890717
17	0,1812934

Tabelle 4.5: Logarithmen der Nullstellen von $x^n - x^{n-1} - 1$.

- Es werden also Pakete mit Übertragungszeit t_0 (unverzögert) und $n \times t_0$ (verzögert) verschickt.
- Die Übertragung von 792 Paketen sollte

  $$T_n := 792 \times (t_0 + \frac{n-1}{2} t_0) = 792 \times \frac{n+1}{2} t_0$$

  Sekunden benötigen, die gemessenen Übertragungszeiten weichen aber davon ab. Anstelle dieses Wertes wird im Folgenden die gefundene Approximation $T'_n = 380(n-1)t_0 + 4,56$ verwendet.
- Man sucht die positive Nullstelle a_n von $x^n - x^{n-1} - 1$ und berechnet deren Logarithmus – das ergibt die Kapazität $C_n = \log_2 a_n$. Die auf sieben Nachkommastellen gekürzten Werte C_n für $n \in \{3, 4, \ldots, 17\}$ zeigt Tabelle 4.5.
- Man teilt die Übertragungszeit durch t_0 und multipliziert mit der Kapazität. Damit erfährt man, wie viele der 792 Zeichen übertragen werden können: $TK_n = T'_n/t_0 * C_n$.
- Schließlich berechnet man $792/TK_n - 1$, um die Überlastung des Kanals zu erhalten.

Man kann die Ausdrücke vereinfachen:

$$TK_n = \frac{T'_n}{t_0} C_n = \frac{380(n-1)t_0 + 4,56}{t_0} C_n = [380(n-1) + \frac{4,56}{t_0}] C_n$$

$$\approx [380(n-1) + 137.3494] C_n$$

und am Ende

$$792/TK_n - 1 = \frac{792}{[380(n-1) + 137.3494] C_n} - 1$$

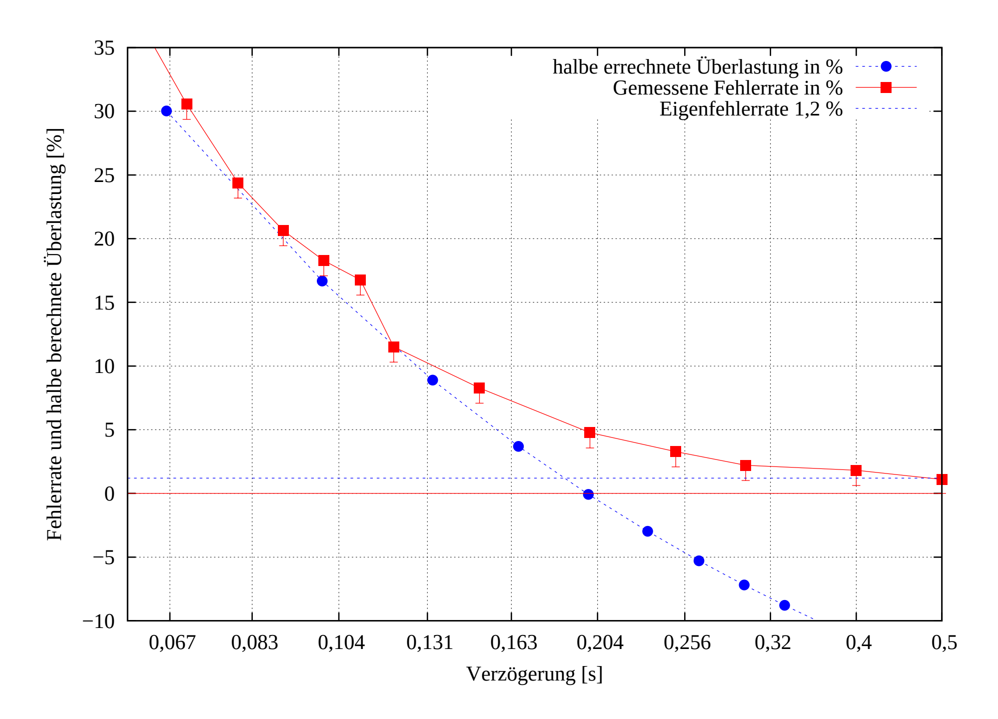


Abbildung 4.21: Gemessene Fehlerrate vs. halbe errechnete Überlastung.

Abbildung 4.21 vergleicht diese Werte mit den gemessenen Fehlerraten. Ab dem Punkt, an dem die berechnete Überlastung 0 unterschreitet, ergeben die Berechnungen keine Überlast mehr, rechtfertigen also mathematisch keine Fehler. In der Abbildung ist die x-Achse logarithmisch dargestellt; die Achsenmarkierungen entstanden durch Multiplikation des oberen Wertes 0,5 mit Potenzen von 0,8.

Die Fehlerbalken an den gemessenen Fehlerraten ziehen von jedem Wert die Eigenfehlerrate von 1,2 % ab.

Die Grafik zeigt zunächst, dass die Fehlermesswerte im erwarteten Bereich liegen, also stets oberhalb der halben berechneten Überlastung. Letztere hat aber nicht den Verlauf einer Exponentialkurve. Durch Experimentieren wurde eine Funktion gefunden, der die meisten Punkte (der berechneten Überlastung) sehr nahe kommen:

$$f(x) = -11,8 \log(5,1x)/x^{0,32} = c_1 \frac{\log(c_2 x)}{\sqrt[25]{x^8}}$$

(für zwei Konstanten c_1, c_2), und auch die Funktion $g(x) = c_1 \log(c_2 x)/\sqrt[3]{x}$ liegt nah an den Werten. Abbildung 4.22 (auf der folgenden Seite) vergleicht die Überlastungen mit den Funktionen f und g.

Die Messwerte bestätigen die theoretischen Ergebnisse von Moskowitz und Miller [MM94], weil die auf dieser Arbeit basierenden Kapazitätsgrenzen nicht überschritten werden; sie werfen aber die Frage auf, warum es ab einer Verzögerung von 0,2 s überhaupt noch Fehler gibt: Für $v > 0,2$ s ist der Kanal nicht überlastet, so dass die Fehlerrate nicht oberhalb der in Kapitel 4.5.5 bestimmten Eigenfehlerrate von 1,2 % liegen sollte. Das ist im Bereich von 0,2 s bis 0,4 s aber der Fall: Die Fehlerraten lagen im zweiten Testlauf zwischen 1,82 % und 4,77 %.

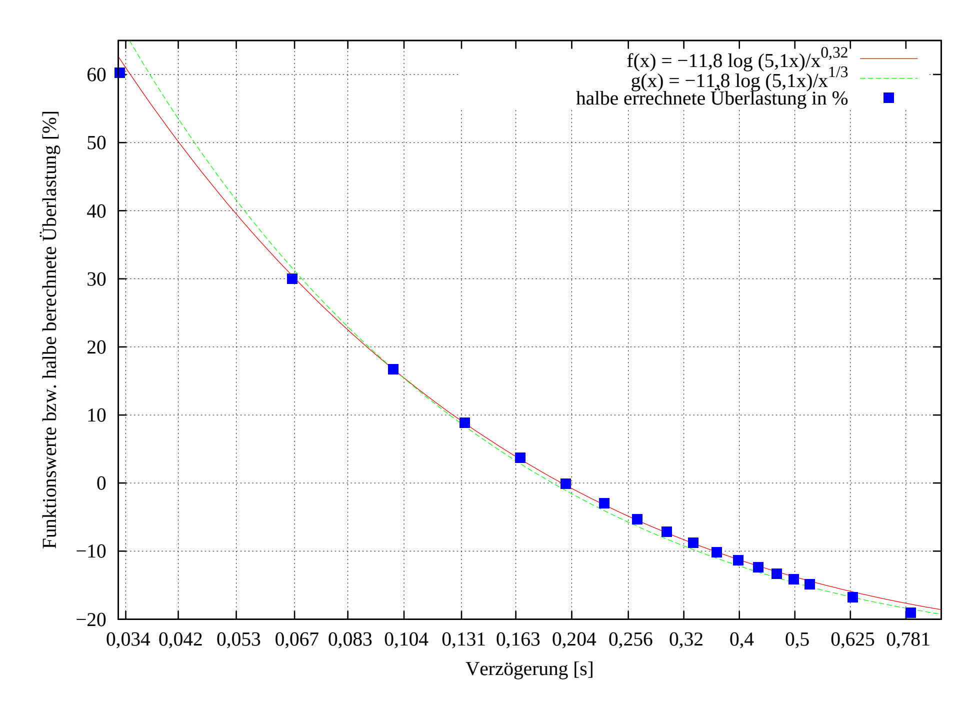


Abbildung 4.22: Approximation der berechneten Überlastungen.

4.9 Beschränkungen

Die Implementierung des verdeckten Kanals weist einige Einschränkungen auf. Die wichtigste ist, dass die Software voraussetzt, dass die komplette (verdeckte) Botschaft im Rahmen eines einzigen Zugriffs auf den Web-Server übertragen wird.

Auf der Server-Seite ist der Daemon-Prozess so ausgelegt, die Nachricht problemlos auf mehrere separate Datenabrufe des Clients aufzuteilen. Auf der Client-Seite sind Anpassungen am Proxy-Server und am Analyseprogramm notwendig, um diese Situation berücksichtigen zu können.

Durch die relative Kapazität von einem Bit pro 4-KByte-Block ist die entwickelte Software nur für die verdeckte Übertragung sehr kleiner Datenmengen geeignet – eine Verbesserung war durch Änderungen des Apache-Quellcodes nicht möglich, da vorgegebene Blockgrößen unter 4 KByte sich nicht auf die Größe der tatsächlich übertragenen Blöcke auswirkten.

Um hier bessere Ergebnisse zu erreichen, könnte man dem Apache-Server einen Proxy vorschalten, der dann die Kommunikation mit dem Daemon-Prozess übernimmt und kleinere Blöcke versendet; alternativ wäre ein tieferer Eingriff in die Apache-Quellen möglich.

4.10 Zusammenfassung

Die Implementierung eines verdeckten Kanals über Verzögerungen einzelner Datenpakete war erfolgreich; eine ständig fehlerfreie Übertragung wurde allerdings nicht erreicht: Im realitätsnächsten Szenario der Kommunikation zwischen zwei Rechnern im Internet blieb selbst bei einer Verzögerung von einer halben Sekunde eine geringe Fehlerrate.

Durch den Einsatz eines Fehlerkorrekturverfahrens konnte die Fehlerquote in vielen Fällen reduziert werden; das allerdings zu dem Preis einer geringeren Kapazität, da pro Zeichen 10 statt 6 Bit übertragen werden mussten. Immerhin waren die Übertragungen bei 0,4 s und 0,5 s Verzögerung im Test zu 100 % fehlerfrei.

Bei diesen hohen Verzögerungen ist die Gefahr groß, dass der verdeckte Kanal beobachtet wird, weil sich die Unterschiede in den Übertragungszeiten deutlich außerhalb der normalen Schwankungen (etwa durch hohe Netzlast) bewegen.

Ein stärkeres Fehlerkorrekturverfahren mit erheblich höherer Redundanz würde kürzere Verzögerungen ermöglichen. Wenn nur sehr geringe Datenmengen übertragen werden sollen, ist dies ein möglicher Weg, die Auffälligkeit des Kanals zu verringern.

Eine exponentielle Abhängigkeit der Fehlerrate von der Verzögerungszeit wird aufgrund der gemessenen Werte vermutet, konnte aber nicht mathematisch erklärt werden.

Im folgenden und letzten Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst und mögliche Erweiterungen vorgestellt.

Kapitel 5

Zusammenfassung und Erweiterungen

Dieses Kapitel fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche Erweiterungen.

5.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein unidirektionaler verdeckter Kanal implementiert, der HTTP-Übertragungen als Trägerkanal verwendet und bei Übertragungen einzelne Pakete zeitlich verzögert: So entsteht ein Zeitkanal.

Im Wesentlichen waren die Erstellung bzw. Modifikation von vier Programmen notwendig:

- Es wurde ein Daemon entwickelt, der die verdeckte Kommunikation steuert und auf dem Rechner betrieben wird, der als Quelle der verdeckten Übertragungen dient.
- Der Apache-Web-Server wurde so modifiziert, dass er vor dem Versand jedes Datenpakets bei obigem Daemon-Prozess nachfragt, ob das Paket verzögert werden soll.
- Für die Empfängerseite wurde ein HTTP-Proxy-Server geschrieben, der als transparenter Proxy Anfragen entgegennimmt und an den Zielrechner weiterleitet, dabei aber zusätzlich die Verzögerungszeiten der einzelnen Pakete misst und in eine Protokolldatei schreibt.
- Die Auswertung der Protokolldaten übernimmt ein Analyseprogramm.

Dieser Kanal kann zur verborgenen Übertragung von Informationen verwendet werden – das setzt allerdings voraus, dass nicht nach Anomalien im Übertragungsverhalten gesucht wird: In Kapitel 4.7 wurde gezeigt, dass sich Übertragungen mit und ohne Verzögerung in ihrer Datendurchsatzstatistik unterscheiden. Bei unverzögerter Übertragung ist der Durchsatz recht einheitlich, während er bei verzögerter Übertragung stark schwankt. Ein mit `ethereal` angezeigtes Datendurchsatzdiagramm zeigt dies deutlich. Zudem steigt bei niedriger Fehlerrate durch die notwendigen Verzögerungen die Übertragungszeit für die Trägerdatei erheblich.

Die Qualität der verdeckten Übertragung über das Internet leidet daran, dass

- einerseits die vom Apache-Server verschickten 4-KByte-Blöcke auf TCP-Ebene fragmentiert (genauer: in kleinere TCP-Segmente aufgeteilt) werden
- und andererseits eine Anpassung der Blockgröße an die TCP-Segmentierung in den Apache-Quellen nicht möglich war.

Die Segmente „passen nicht" zur Blockgröße: Im Test wurden TCP-Pakete der Größe 1420 mit enthaltenen HTTP-Paketen der Größe 1388 übertragen. Bei sehr hoher Verzögerung aller Pakete würde dies zur Aufteilung jedes 4-KByte-Blocks in drei Pakete mit HTTP-Anteilen von 1388, 1388 und 1320 führen. Gelegentlich geschieht auch genau das, in vielen Fällen werden aber das Ende eines Blocks und der Anfang des folgenden Blocks vom TCP-Stack zu einem Segment zusammengefasst. Dadurch werden die Blockgrenzen fließend, und auf Empfängerseite werden verzerrte Verzögerungszeiten gemessen.

Bei Übertragungen über das Internet ist dieser Effekt stärker als im lokalen Netz, obwohl auch dort die Blöcke vom TCP-Stack zerlegt werden. Daher waren im Internet-Szenario höhere Verzögerungszeiten nötig, um eine akzeptable Fehlerrate zu erzielen.

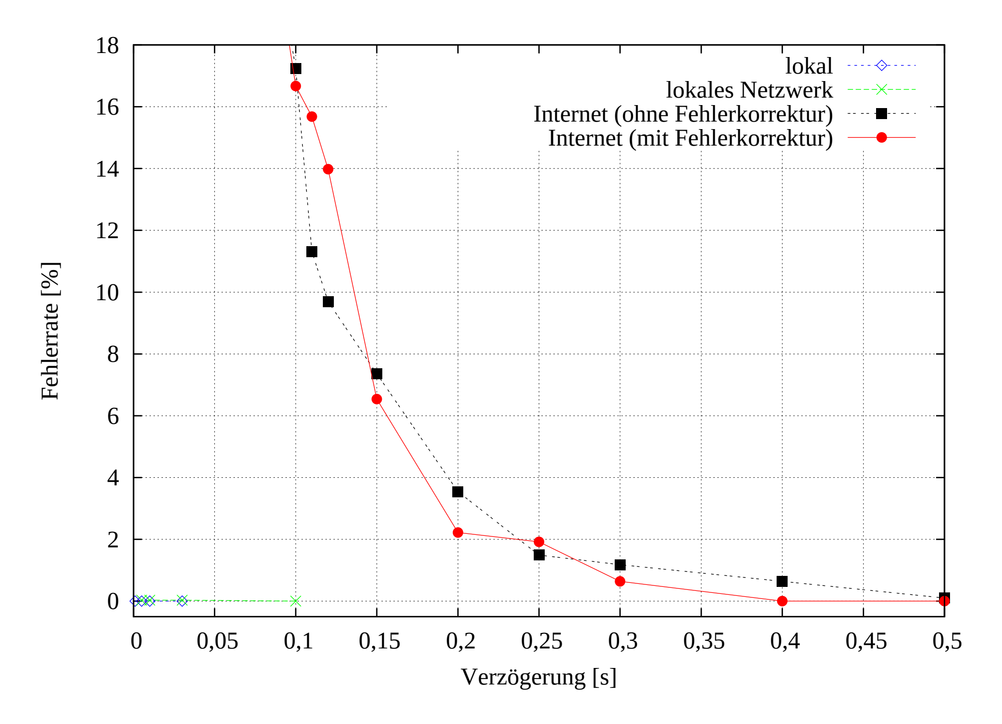


Abbildung 5.1: Fehlerraten für verschiedene Testszenarien und Verzögerungen.

Die Messwerte beim Transfer über das Internet legen die Annahme nahe, dass die Fehlerrate mit sinkender Verzögerungszeit exponentiell steigt – dieser Zusammenhang konnte allerdings nicht mathematisch gestützt werden. Beim Vergleich mit

theoretischen Ergebnissen in Kapitel 4.8 wurde aber festgestellt, dass die dort gefundenen Schranken für die Kapazität nicht überschritten werden.

Die relative Kapazität von 1 Bit pro 4096 Byte Trägerdaten ist gering, reicht aber für die Übertragung geringer Datenmengen aus. So benötigte beispielsweise die für den Test generierte verdeckte Nachricht mit 128 Zeichen (aus einem auf 6 Bit reduzierten Zeichensatz) eine knapp über 3 MByte große Trägerdatei.

Als Fehlerkorrekturverfahren wurde ein einfacher Hamming-Code verwendet, der 6-Bit-Blöcke durch Anhängen von vier Paritätsbits in 10-Bit-Blöcke kodiert. Sein Einsatz verbesserte die Ergebnisse nicht wesentlich, teilweise war die Benutzung sogar schädlich.

Die Belastung des Web-Servers durch parallele Zugriffe änderte die Übertragungsqualität des verdeckten Kanals nicht.

5.2 Erweiterungen

Die Arbeit kann auf verschiedene Weisen ausgebaut werden und damit den Nutzen der vorgestellten Implementierung erhöhen.

5.2.1 Rückkanal

Da die Übertragungen auf dem verdeckten Kanal im Internet-Szenario stark fehlerbehaftet sind, ist die Integration eines Rückkanals hilfreich, über den der Client den korrekten Empfang eines Blocks bestätigen oder verneinen kann.

Auch für diesen Rückkanal ist der Einsatz zeitlicher Verzögerungen denkbar: Nach jedem empfangenen Block (mit Prüfsummen) überprüft der Client-Proxy die Daten und verzögert im Fall einer fehlerhaften Übertragung die Anforderung des nächsten Pakets.

Der Daemon auf Server-Seite merkt sich die Abstände zwischen den letzten Anforderungen von diesem Rechner und interpretiert eine Verzögerung so, dass er den letzten Block neu übertragen soll.

Eine Schwierigkeit in diesem Modell ist, dass der Daemon keine Möglichkeit hat, eine statistische Analyse der vollständigen Kommunikation durchzuführen.

5.2.2 Integration weiterer Verfahren

Die verdeckte Kommunikation kann mit Verschlüsselung und/oder Steganographie verknüpft werden. So wäre es etwa denkbar, Web-Seiten mit Bildern zu übertragen und in den Bildern mit steganographischen Methoden einen Schlüssel zu verstecken, der zur Entschlüsselung der kodierten Daten auf dem verdeckten Kanal genutzt wird.

Soll ein größeres Datenvolumen übermittelt werden und ist der verdeckte Kanal zu langsam, ist auch der umgekehrte Weg denkbar: Der Schlüssel wird über den verdeckten Kanal übertragen, und die kodierten Daten sind im Bild verborgen.

Solche Ergänzungen führen allerdings dazu, dass die Daten des regulären Kanals modifiziert werden.

5.2.3 Interaktivität

Die Analyse findet erst nach vollständig erfolgter Übertragung der Trägerdatei statt. Es sind Situationen denkbar, in denen die Auswertung unmittelbar erfolgen soll.

Zu diesem Zweck könnte jeder Nachricht ein zusätzlicher „Kalibrierungsblock“ vorangestellt werden, dessen Inhalt dem Analyseprogramm bekannt ist und das ihm hilft, durchschnittliche (erwartete) Verzögerungszeiten für den Rest der Nachricht zu berechnen. Darauf basierend wäre dann eine sofortige Ausgabe der folgenden Nachrichtenbits möglich.

5.2.4 Unterstützung mehrerer Trägerdateien

In der implementierten Version wird vorausgesetzt, dass die gesamte verdeckte Nachricht mit Hilfe des Downloads einer ausreichend großen Trägerdatei übermittelt wird. Üblicher ist aber das Lesen vieler kleinerer Web-Seiten, über die sich dann nur partielle Informationen übertragen lassen.

Eine Erweiterungsmöglichkeit liegt damit in der Verwendung mehrerer Trägerdateien. Zu diesem Zweck könnte man wie folgt vorgehen:

- Der Apache-Server überträgt neben der IP-Adresse auch den Dateinamen der vom Client angeforderten Datei an den Daemon.
- Der Daemon registriert den Wechsel der Datei und baut einen neuen Start-Header in die noch vorhandene Teilnachricht ein – dieser Start-Header hilft dem Analyseprogramm auf der Client-Seite dabei, die unterteilten Daten zusammenzusetzen.
- Das Analyseprogramm wird so angepasst, dass es die Protokolldatei des Proxys nach mehreren Blöcken durchsucht.

5.2.5 Entwicklung einer grafischen Oberfläche

Für die Steuerung des Daemons könnte eine grafische Oberfläche entwickelt werden, die eine komfortablere Verwaltung der Kommunikationspartner und Nachrichten erlaubt sowie durch regelmäßige Statusabfragen einen besseren Überblick über den Fortschritt der verdeckten Übertragungen gibt.

Erste Versuche mit Python und Gtk (mit Hilfe des *Gtk User Interface Builders* GLADE [Gla]) sind Erfolg versprechend verlaufen.

5.2.6 Ausnutzung der TCP-Segmentierung

Durch größere Änderungen an den Techniken, die der Apache-Server zum Versand der Daten verwendet, könnte die für die Verzögerungen genutzte Blockgröße an die maximale Segmentgröße (MSS) der jeweiligen Verbindung angepasst werden. Bei einer typischen MSS von 1460 Byte würde dies die relative Kapazität des verdeckten Kanals ungefähr verdreifachen ($4096/1460 \approx 0{,}356$).

Ist dann noch das Timing-Verhalten des TCP-Stacks bekannt, ist eine Feineinstellung möglich.

Indem der Server genau passende Segmente erzeugt, ist dann auch das Problem der Auffälligkeit unterschiedlicher Paketgrößen behoben.

Kapitel 6

Andere Arbeiten zu verdeckten Kanälen

Im letzten Kapitel werden weitere Arbeiten vorgestellt, die sich mit der Implementierung verdeckter Kanäle beschäftigen.

Beim Vergleich verschiedener Ansätze fällt als Erstes auf, dass die meisten Arbeiten Verfahren einsetzen, welche die (auf dem Trägerkanal verschickten) Daten modifizieren. Der verdeckte Kanal, der in dieser Arbeit entwickelt wurde, verzichtet darauf.

In allen Fällen werden aber verdeckte Kanäle im Sinne der Definition aus Kapitel 2.2 erzeugt, nach der ein Kanal verdeckt ist, wenn über ihn in unerwarteter Weise kommuniziert wird.

6.1 MIX-Netze

MIX-Netze, wie sie z. B. von Chaum [Cha81] beschrieben werden, erlauben die Verschleierung von Absender- und Empfängerinformationen und damit anonymes Versenden von Nachrichten. Anders als bei der einfachen Verschlüsselung von Nachrichten mit *Public-Key*-Kryptographie (z. B. PGP, GPG) wird nicht nur die Nachricht für Fremde unlesbar, sondern es werden auch Quelle und Ziel verschleiert, da eine große Anzahl Parteien parallel alle Informationen untereinander austauschen – also auch nicht Beteiligte die Nachricht erhalten (und verwerfen). In der stärksten Form kann nicht einmal festgestellt werden, *ob* überhaupt Kommunikation stattfindet: Dann schickt das Netzwerk ständig Dummy-Nachrichten, die ohne Schlüssel nicht von richtigen Nachrichten unterscheidbar sind. Alle empfangenen Dummy-Nachrichten werden (wie die für andere Sender bestimmten) verworfen.

Diesen Ansatz greift die Arbeit von Bauer [Bau03] auf. Im vorgeschlagenen Verfahren werden die WWW-Clients unbeteiligter Rechner für den Datentransport verwendet. Ein Sender überträgt seine Nachricht (*public-key*-verschlüsselt) an einen der MIX-Knoten, dieser verteilt die Nachricht nach folgendem Verfahren an andere Knoten weiter (Abbildung 6.1):

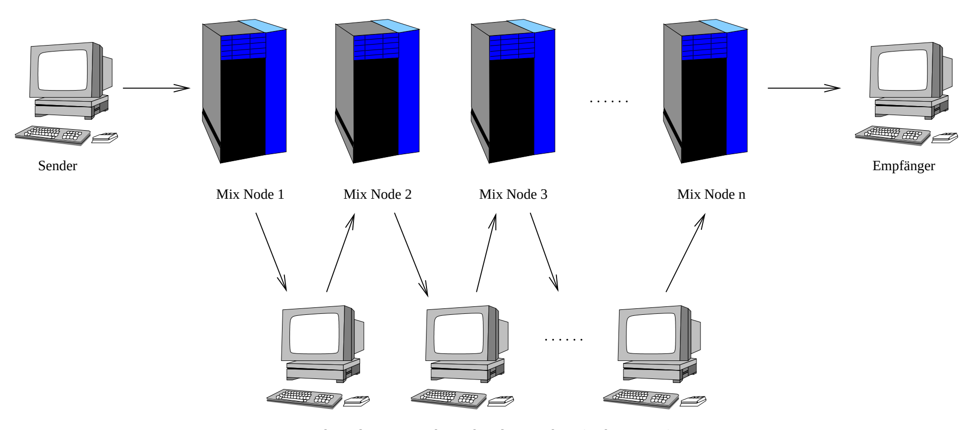


Abbildung 6.1: MIX-Netzwerk mit unbeteiligten Web-Clients.

- Beim Abruf einer Web-Seite vom MIX-Knoten werden neben den eigentlichen Inhalten Datenpakete mit Zieladressen an den Client übertragen. Für die Weiterleitung werden *Redirects* (Umleitungen), Cookies, *Referer*-Header, die HTML-Elemente `frame`, `iframe`, `script`, `link` etc. und aktive Inhalte (Java, JavaScript etc.) verwendet.
- Der Client interpretiert diese Informationen. Dadurch wird Kontakt zu einem anderen MIX-Knoten aufgebaut und die Nachricht weitergeleitet.
- Über dieses Verfahren landet die Nachricht schließlich bei einem MIX-Zielknoten, von dem der vorgesehene Empfänger das Paket abholt.

Der Weg der eigentlichen Nachrichten kann nicht beobachtet werden, und die Methode der Kommunikation verschleiert die Identität der Kommunikationspartner. Insbesondere ist es nicht möglich, Sender und Empfänger von den unbeteiligten Transport-Clients zu unterscheiden.

Da die MIX-Knoten grundsätzlich neben den Nutzdaten (dem Inhalt der Web-Seite) MIX-Daten übertragen, kann nicht festgestellt werden, ob verdeckt kommuniziert wird oder nicht – es fällt lediglich auf, dass die Zusatzinformationen bei jedem Abruf der Web-Seite wechseln.

Ähnlich wie bei dieser Diplomarbeit ist das Ziel der MIX-Netze, die Tatsache zu verbergen, dass Kommunikation stattfindet. Zur Verschleierung werden Daten auch an viele unbeteiligte Clients übertragen. Fällt bei einer Analyse auf, dass ein MIX-Netz betrieben wird, bleiben dennoch Sender und Empfänger einzelner Nachrichten unbekannt. Da Informationen sichtbar im Header oder Body der HTTP-Daten abgelegt werden, unterscheidet sich das MIX-Konzept im Punkt der Datenveränderung von dieser Arbeit.

6.2 Verdeckte Kanäle über HTTP

In einer jüngeren Arbeit [DC03] verwenden die Autoren das HTTP-Protokoll, um einen verdeckten Kanal zu etablieren. Es wird das Tunneln von Daten über HTTP

beschrieben und demonstriert, wie beliebige Daten oder Netzwerkdienste über HTTP-Tunnel übertragen werden können, wenn alle anderen Ports gesperrt sind. Die hier übertragenen Daten sind nicht in der Weise verborgen, wie es beim in dieser Arbeit implementierten verdeckten Kanal der Fall ist, der auf die Veränderung der Trägerdaten verzichtet.

Die Arbeit beschreibt verschiedene Ansätze, Daten im HTTP-Protokoll unterzubringen. Dazu kommen Sicherheitsaspekte (Verschlüsselung, Authentifizierung, Datenintegrität) und steganographische Methoden.

Eine Implementation der beschriebenen Techniken ist auf der Cctt-Homepage [Cct] erhältlich.

6.3 LOKI2

Das Projekt LOKI, benannt nach Loki, dem Gott der Lügen und Täuschung, basiert darauf, dass Ping-Pakete (ICMP Ping, RFC 792 [Pos81a]) von den meisten Firewalls als unproblematisch betrachtet werden.

LOKIs Software nutzt die Tatsache, dass ICMP-ECHO- und ICMP-ECHOREPLY-Pakete ungenutzte Datenblöcke enthalten, für die RFC 792 nur vorschreibt, dass empfangene Datenblöcke unverändert an den Absender zurück geschickt werden müssen.

Die Grundlagen und die Implementierung sind in zwei Arbeiten [Dae96, Dae97] beschrieben. Auch eine Arbeit von Goltz [Gol03] untersucht das LOKI-Projekt und stellt fest, dass Kommunikation über LOKI anhand von Auffälligkeiten der Sequenznummern in den Ping-Paketen entdeckt werden kann.

Der erzeugte Kanal ist in dem Sinne verdeckt, dass die Nutzung von Ping-Paketen für die Datenübertragung unerwartet ist. Durch die Verwendung eines üblicherweise ungenutzten Datenblocks (also die Veränderung von Nutzdaten) unterscheidet er sich vom Ansatz dieser Diplomarbeit.

6.4 Infranet

Infranet [FBH+02] ist ein HTTP-Tunnel mit der speziellen Zielsetzung, Internet-Zensur zu umgehen.

Teilnehmende Clients (denen eine Liste von Infranet-Servern bekannt sein muss) kodieren eine Anfrage-URL in eine Liste von normalen Anfragen an den Infranet-Server. Der Server wertet die Reihenfolge dieser Anfragen aus und ermittelt daraus die gewünschte URL. Er lädt die angeforderten Informationen herunter und überträgt sie in seinen Antworten auf weitere Anfragen vom gleichen Client. Die Daten werden dazu über Steganographie in JPG-Bilder eingebettet.

Während es viele HTTP-Tunnel-Programme gibt, nennt Infranet besondere Ziele, die in einer Zensurumgebung wichtig sind:

- Es darf dem Client nicht nachzuweisen sein, dass er eine Anfrage an das Infranet gestellt hat. Das ist beispielsweise dann wichtig, wenn eine Umgehung der Zensurmechanismen strafbar ist.
- Über den Ausschluss der Nachweisbarkeit hinaus soll die Kommunikation statistisch unauffällig sein – die von Infranet ausgelösten Datentransfers sollen also wie normales Surf-Verhalten aussehen.
- Infranet-Server sollen nicht erkennbar sein, damit Zensureinrichtungen den Zugriff auf sie nicht sperren.

Infranet unterscheidet sich – abgesehen von der viel größeren Komplexität – in zwei Punkten von dieser Arbeit: Zum einen findet durch die Verwendung von Steganographie wieder ein Eingriff in die Trägerdaten statt, zum anderen ergeben sich aus der Zielsetzung, Zensur zu umgehen, andere Anforderungen, wie etwa, die Entdeckung des Kanals in jedem Fall auszuschließen.

Anhang A

Aufgabenstellung

Dieser Anhang gibt den Originalwortlaut der Aufgabenstellung wieder, Quelle: http://www-i4.informatik.rwth-aachen.de/lufg/.

Ausnutzung verdeckter Kanäle am Beispiel eines Webservers

(Practical Experiences with Covert Channels using a Webserver)

Vorgeschichte/Background

In einem Krankenhausprojekt wurde die elektronische Krankenakte eingeführt: Patienten und Ärzte hatten nun gleichermaßen die Möglichkeit, Diagnosen, Röntgenbilder und Laborergebnisse jederzeit und überall einzusehen. Natürlich war dies nur den Patienten selbst und den behandelnden Ärzten erlaubt; durch Verwendung eines Zugriffskontrollmechanismus wurde unbefugten Personen der Zugriff auf die Daten verweigert. Die neuen Technologien erzeugten aber auch neue Probleme. So konnten Patienten beispielsweise sofort nach der Labordiagnose (und zum Teil vor den Ärzten) auf die Resultate zugreifen. Dies führte in drastischen Fällen (z. B. bei einer Krebsdiagnose) zu Schocks und Kreislaufzusammenbrüchen unter den Patienten. Um dies zu vermeiden, wurde der Zugriffskontrollmechanismus derart geändert, dass Patienten der Zugang zu ihren eigenen Daten verweigert wurde, falls eine tödliche Diagnose vorlag. Daraufhin häufte sich allerdings die Anzahl der besorgten Patienten, da jede Zugriffsverweigerung (z. B. aufgrund eines versehentlich falsch eingegebenen Passwortes) sofort mit einer tödlichen Diagnose in Verbindung gebracht wurde.

Aufgabenstellung/Task

Wenn Informationen über einen Kanal fließen, ohne dass sie fließen sollen, spricht man von einem verdeckten Kanal (covert channel). Im obigen Krankenhausbeispiel existiert ein verdeckter Kanal mit Informationen, ob der Patient eine tödliche Diagnose hat oder nicht: aus dem Ausbleiben von Informationen (Zugriffsverweigerung

bei korrekter Anmeldung) kann bereits Information gewonnen werden. Verdeckte Kanäle treten aber auch in vielen anderen Situationen auf und sind ein großes Problem in Hochsicherheitssystemen, wie z. B. militärischen IT-Systemen. Verdeckte Kanäle werden auch bevorzugt von Trojanischen Pferden zur geheimen Kommunikation mit einem Angreifer verwendet.

In dieser Diplomarbeit geht es darum, praktische Erfahrungen mit verdeckten Kanälen zu sammeln. Es sollen Daten unerkannt von einem Rechner (Server) zu einem anderen Rechner (Client) über einen Standard-HTTP-Kanal übertragen werden. Ziel ist es, durch Ausnutzen von Zeit-, Kanal- und Protokollredundanz, so viele Informationen wie möglich auf einen HTTP-Kanal aufzuspielen und wieder auszulesen. Technisch soll dies durch Veränderung eines Standard-Webservers (Apache) und der Modifikation des Proxys auf Client-Seite realisiert werden.

Vorausgesetzt werden Erfahrungen in der Installation und Wartung eines Webservers, Kenntnisse in HTTP und Spaß am Experimentieren.

Anhang B

Benutzerhandbuch

Dieser Anhang erklärt Installation und Handhabung der Software, die im Rahmen der Diplomarbeit implementiert wurde.

B.1 Installationsanleitung

Die Testumgebung besteht aus zwei Komponenten, der Server- und der Client-Seite. Alle nötigen Programme finden Sie im Internet unter `http://privat.hgesser.com/docs/Info-Diplom/` und auf der beigelegten CD.

B.1.1 Einrichten des Apache-Servers

1. Entpacken Sie die Apache-Quellen (`apache_1.3.31.tar.gz`), die auf der Begleit-CD im Verzeichnis `software/apache/` liegen.
2. Wenden Sie den Patch `apache_1.3.31-cc-hge.patch` (ebenfalls aus dem Verzeichnis `software/apache/`) mit

   ```
   patch -p0 < apache_1.3.31-cc-hge.patch
   ```

   auf die Apache-Quellen an.
3. Wechseln Sie in das Apache-Verzeichnis und kompilieren und installieren Sie die gepatchte Version mit

   ```
   cd apache_1.3.31
   ./configure --prefix=PFAD
   make
   make install
   ```

   wobei `PFAD` durch das gewünschte Installationsverzeichnis zu ersetzen ist – das kann auch ein Unterverzeichnis Ihres Home-Verzeichnisses sein.
4. Kopieren Sie den Daemon `ccdaemon.py` (aus dem Verzeichnis `software/daemon/`) in ein geeignetes Verzeichnis `BIN` (zum Beispiel `~/bin/`) und starten Sie ihn mit `BIN/ccdaemon.py &`.

5. Prüfen Sie mit

   ```
   echo "C del 1" | netcat localhost 4999
   ```

   ob der Daemon erreichbar ist – Sie sollten darauf die Fehlermeldung „Error: IP address 1 not found“ erhalten.

6. Erzeugen Sie eine Apache-Konfigurationsdatei bzw. nehmen Sie Änderungen an der mitgelieferten Konfigurationsdatei vor. Details dazu finden sich in der Apache-Dokumentation.

7. Starten Sie dann den Apache-Server mit `PFAD/bin/apachectl start`.

B.1.2 Einrichten des Client-Proxys

1. Kopieren Sie auf dem Client-Computer den Client-Proxy `ccproxy.py` (aus dem CD-Verzeichnis `software/proxy`) in ein geeignetes Verzeichnis `BIN`, z. B. `~/bin/`.

2. Passen Sie im Programm die Variable `LOGFILE` an: Sie bestimmt, in welche Datei die Messwerte geschrieben werden.

3. Starten Sie den Proxy mit `BIN/ccproxy.py &`.

4. Konfigurieren Sie den Web-Browser, mit dem Sie testen wollen. Als Proxy-Server stellen Sie `localhost` und als Port 21000 ein. Wenn Sie `wget` einsetzen, definieren Sie den Proxy über eine Umgebungsvariable:

   ```
   export http_proxy="http://localhost:21000"
   ```

   Beim Aufruf von `wget` ist jeweils die Option `--proxy=on` anzugeben.

B.1.3 Einrichten des Analyseprogramms

Kopieren Sie das Analyseprogramm `ccanalyse.py` aus dem CD-Verzeichnis `software/analyse/` in ein geeignetes Verzeichnis, z. B. `~/bin/`.

B.2 Anpassungen

B.2.1 Änderung der Ports

Folgende Ports werden von den an der Kommunikation beteiligten Komponenten standardmäßg verwendet:

- 80 (HTTP): Apache-Server
- 4999: Daemon
- 21000: Proxy

Wenn einer oder mehrere dieser Ports auf Ihren Systemen bereits belegt sind, können Sie die Ports ändern. Dazu sind die folgenden Schritte notwendig:

- **Daemon:** Der Port, auf dem der Daemon Anfragen entgegennimmt, wird in `ccdaemon.py` über die Zeile

  ```
  CC_PORT = 4999
  ```

  festgelegt. Ändern Sie diesen Wert und beenden und starten Sie den Daemon neu. Zusätzlich muss das Kontrollprogramm `ccctrl` angepasst werden, das Befehle mit `netcat` an den Daemon schickt: Auch dort ist der neue Port einzutragen.

- **Proxy:** Der Proxy verwendet standardmäßig Port 21000. Um diesen zu ändern, suchen Sie in der Datei `ccproxy.py` nach der Zeile

  ```
  s.bind(("",21000))                 # An Port 21000 binden
  ```

  Ändern Sie im `bind`-Aufruf den Wert 21000 und beenden und starten Sie den Proxy neu.

- **Apache-Server:** Um den Standard-Port 80 des Apache-Servers zu verändern, bearbeiten Sie dessen Konfigurationsdatei (oft: `httpd.conf`). Suchen Sie darin nach der Zeile

  ```
  Port 80
  ```

  Ändern Sie den Wert, speichern Sie die Datei und starten Sie den Apache-Server neu.

B.2.2 Änderung der Verzögerungszeit

Die Verzögerungszeit für den Apache-Server wird in der Datei `src/include/covert_channel.h` im Apache-Quellcode-Verzeichnis festgelegt. Um die Zeit zu ändern, passen Sie die Zeile

```
#define AP_CC_SLEEPTIME 0.001
```

an. Die Zeit wird in Sekunden angegeben.

Nach der Änderung muss der Apache-Server mit

```
make && make install
```

neu installiert und neu gestartet werden.

B.2.3 Aktivieren der Fehlerkorrektur

Die Fehlerkorrektur muss an zwei Stellen ein- bzw. ausgeschaltet werden:

- Im Daemon `ccdaemon.py` ist die Zeile `ERROR_CORRECTION = 0` anzupassen: Der Wert `0` steht für deaktivierte Fehlerkorrektur, 1 schaltet die Korrektur ein. Danach ist ein Neustart des Daemons notwendig. Nach dem Abbruch des Programms fällt eine kurze Wartezeit an, da der von `ccdaemon.py` belegte Port noch nicht freigegeben wurde.
- Auf Empfängerseite ist das Analyseprogramm `ccanalyse.py` betroffen. Es enthält eine gleichlautende Zeile, die entsprechend anzupassen ist.

B.3 Benutzung

In diesem Abschnitt wird davon ausgegangen, dass Sie bereits auf Server-Seite den Apache-Server und den Daemon sowie auf Client-Seite den Proxy gestartet haben, wie es im vorangegangenen Abschnitt beschrieben wurde. Damit ist das System bereit zum Versand verdeckter Nachrichten.

B.3.1 Kontrollprogramm

Das Kontrollprogramm `ccctrl` kennt die folgenden Befehle:

- `ccctrl add` *`ip`*: IP-Adresse hinzufügen
- `ccctrl msg` *`ip Nachricht`*: Nachricht in die FIFO-Liste für die angegebene IP-Adresse eintragen
- `ccctrl msgfile` *`ip Dateiname`*: Nachricht aus Datei lesen und in die FIFO-Liste für die angegebene IP-Adresse eintragen
- `ccctrl stat` *`ip`*: Status zur IP-Adresse ausgeben
- `ccctrl allstat`: Alle Statusinformationen ausgeben
- `ccctrl del` *`ip`*: IP-Adresse entfernen

Eine Beispielsequenz, die einen neuen Kommunikationspartner 1.2.3.4 hinzufügt, für diesen eine Nachricht einträgt und nach kurzer Wartezeit den Status abfragt, ist:

```
ccctrl add 1.2.3.4
ccctrl msg 1.2.3.4 Kleine Beispielnachricht
ccctrl allstat
```

B.3.2 Daemon über `netcat` ansteuern

Eine Alternative zur Verwendung von `ccctrl` ist die direkte Eingabe der `netcat`-Aufrufe. (Auch `ccctrl` verwendet diese Methode und ist damit nur eine Arbeitserleichterung.)

- `echo C add` *`ip`* `| netcat` *`server port`*: IP-Adresse hinzufügen
- `echo C msg` *`ip Nachricht`* `| netcat` *`server port`*: Nachricht in die FIFO-Liste für die angegebene IP-Adresse eintragen
- `echo C msgfile` *`ip Dateiname`* `| netcat` *`server port`*: Nachricht aus Datei lesen und in die FIFO-Liste für die angegebene IP-Adresse eintragen
- `echo C stat` *`ip`* `| netcat` *`server port`*: Status zur IP-Adresse ausgeben
- `echo C allstat | netcat` *`server port`*: Alle Statusinformationen ausgeben
- `echo C del` *`ip`* `| netcat` *`server port`*: IP-Adresse entfernen

Welche der beiden Methoden (`ccctrl` oder manuelle `netcat`-Aufrufe) Sie verwenden, ergibt keinen Unterschied.

B.3.3 Das Analyse-Tool

Das Analyseprogramm `ccanalyse.py` kennt keine Optionen und wird einfach aus dem Verzeichnis heraus aufgerufen, in dem sich die Protokolldatei `myproxy.log` des Proxys befindet.

Um die Analyse zu erleichtern, wurde das Skript `process` erstellt:

```
#!/bin/bash
tail -n2 myproxy.log | head -n1 | cut -b 27- > data.in
in2out.py
plot
ccanalyse.py > analyse.txt
```

Die hier abgedruckte Fassung setzt voraus, dass sich `ccanalyse.py` und die beiden Hilfsprogramme `in2out.py` und `plot` im Pfad befinden; anderenfalls muss der volle Pfad zu den Programmen eingetragen werden.

Im Ergebnis erhält man eine Datei `analyse.txt`, die die gefundenen Ergebnisse darstellt; zudem erzeugt `plot` eine Datei `output.eps`, welche die Verzögerungszeiten grafisch visualisiert. Je nach Verzögerungszeit muss in `plot` der Wert `yrange` angepasst werden. `plot` setzt ferner voraus, dass `gnuplot` installiert ist und im Verzeichnis `/usr/local/bin` liegt. Der Wert kann in `plot` angepasst werden. Ist `gnuplot` nicht installiert, kann der `plot`-Aufruf in `process` auskommentiert werden.

Die Dateien `data.in` und `data.out` können nach dem Aufruf gelöscht werden. `myproxy.log` kann für weitere Durchläufe unverändert bleiben, da sich `process` immer die Messdaten aus dem letzten Durchlauf heraussucht.

Die Datei `analyse.txt` hat immer den folgenden Aufbau:

```
Gesamtwerte
Median:     0.0330700000
Mittelwert: 0.1644174141
Abweichung: 0.2013571099
Unterer Block
Minimum:    0.0221800000
Maximum:    0.0330700000
Mittelwert: 0.0323505928
Abweichung: 0.0010117567
Oberer Block
Minimum:    0.0330800000
Maximum:    0.7664200000
Mittelwert: 0.2968975900
Abweichung: 0.2148879426
AUFRAEUMEN
Unterer Block
Minimum:    0.0221800000
Maximum:    0.1620800000
Mittelwert: 0.0365218037
Abweichung: 0.0117027090
Oberer Block
Minimum:    0.0330800000
Maximum:    0.7664200000
Mittelwert: 0.3885785376
Abweichung: 0.1800895766
Nachricht:  111111000000010000100000000000000111000110010111010111000011100111\
0011010011001101001010001111101100100010100011100110011000111011011111010001\
1011001101111010000111001111010110110000110010101101111110110000011000110010 1\
[....]
Nachricht:  010000100000000000000111000110010111010111000011100111001101001100\
1101001010001111101100100010100011100110011000111011011111010001101100110111\
1010000111001111010110110000110010101101111110110000011000110010101110111101 0\
[....]
Text:       O@ .,NN'.:9I1]D4<S';]&S>ASUL,K?L&,KO5'\SO<TOY"W5SGR6]D*;P+9&I@&AO\
2 =UA*VH>!]R%M.GP8Q)P@..ZO-SMU:$HT/Z7/!E$ "Y%.3HM^2#**5QS"*21(N _
Original:   O@ .,NN'.:9I1]D4<S';]&S>ASUL,K?L&,KO5'\SO<TOY"W5SGR6]D*;P+9&I@&AO\
2 =UA*VH>!]R%M.GP8Q)P@..ZO-SMU:$HT/Z7/!E$ *Y%.3HM^2#**5QS"*21(N _
Bitfehler:  1 :  [644]
Laenge:     780
Fehler %:   0.128205128205
```

Neben den statistischen Daten im oberen Bereich sind vor allem die Angaben zur Anzahl der Bitfehler und zur Fehlerquote wichtig.

Anhang C

Kommentierter Programm-Quellcode

Der Anhang enthält den kommentierten Abdruck der wichtigsten Programmteile, die im Rahmen dieser Diplomarbeit erstellt wurden.

C.1 Der Daemon ccdaemon

Das Herzstück auf der Server-Seite ist der Daemon ccdaemon, der die verdeckt zu versendenden Nachrichten verwaltet und bitweise an den modifizierten Apache-Server überträgt.

```
#!/usr/bin/env python
# -*- coding: iso-8859-1 -*-

# cc daemon
# Teil der Informatik-Diplomarbeit von Hans-Georg Eßer
# http://privat.hgesser.com/docs/Info-Diplom/
# (c) 2004, 2005 Hans-Georg Eßer


### Einstellungen
```

Am Anfang werden zwei Parameter festgelegt: CC_PORT enthält den Port, auf dem der Daemon auf Anfragen hört. Die Variable ERROR_CORRECTION legt fest, ob das System mit Fehlerkorrektur arbeiten soll.

```
# CC_PORT: Port, auf dem der Daemon Anfragen entgegen nimmt
CC_PORT = 4999
true = 1
false = 0
ERROR_CORRECTION = 1       # 0: Fehlerkorrektur aus; 1: an
```

Arbeitet der Daemon mit Fehlerkorrektur, ist die Anzahl der übertragenen Bits pro Zeichen der Nachricht 6 oder 10; die Fehlerkorrekturfunktion ergänzt die sechs Bits um vier Paritätsbits.

```
if ERROR_CORRECTION == 1:  # wie viele Bits pro Zeichen?
    CHARLENGTH = 10
else:
    CHARLENGTH = 6
```

```
import SocketServer, time, sys
from string import upper
# sys.stderr = sys.stdout

# Globale Variablen
```

Die globale Variable `comlist` speichert die Informationen über alle Kommunikationspartner des Daemons. Als *Dictionary* (wörtlich: Wörterbuch) legt sie für jede IP-Adresse einen Eintrag ein, der später über `comlist[ip]` abrufbar ist.

```
comlist = {}     # leeres Dictionary
```

C.1.1 Fehlerkorrektur

Die folgenden Code-Zeilen implementieren Kodierung und Dekodierung der Fehlerkorrektur. Sie setzen den Algorithmus um, der in Kapitel 2.7 beschrieben wurde: Auf Basis der Paritätsvektoren $(0, 0, 0, 0, 1, 1)$, $(0, 1, 1, 1, 0, 0)$, $(1, 0, 1, 1, 0, 1)$ und $(1, 1, 0, 1, 1, 0)$ wird ein linearer Code erzeugt, der 6-Bit-Vektoren auf 10-Bit-Vektoren abbildet. Das *Dictionary* `hamming` speichert diese Abbildung h; in `codewords` werden nur die Code-Worte, also Bild(h), abgelegt.

```
### hamming
### Dieses Dictionary speichert die Fehlererkennungskodierung.
### Die Zuordnung über ein Dictionary ist schneller als die
### manuelle Berechnung der Prüfsummen aus den Zeichenketten.

hamming = {
    "000000" : "0000000000", "000001" : "0000011010", "000010" : "0000101001",
    "000011" : "0000110011", "000100" : "0001000111", "000101" : "0001011101",
    "000110" : "0001101110", "000111" : "0001110100", "001000" : "0010000110",
    "001001" : "0010011100", "001010" : "0010101111", "001011" : "0010110101",
    "001100" : "0011000001", "001101" : "0011011011", "001110" : "0011101000",
    "001111" : "0011110010", "010000" : "0100000101", "010001" : "0100011111",
    "010010" : "0100101100", "010011" : "0100110110", "010100" : "0101000010",
    "010101" : "0101011000", "010110" : "0101101011", "010111" : "0101110001",
    "011000" : "0110000011", "011001" : "0110011001", "011010" : "0110101010",
    "011011" : "0110110000", "011100" : "0111000100", "011101" : "0111011110",
    "011110" : "0111101101", "011111" : "0111110111", "100000" : "1000000011",
    "100001" : "1000011001", "100010" : "1000101010", "100011" : "1000110000",
    "100100" : "1001000100", "100101" : "1001011110", "100110" : "1001101101",
    "100111" : "1001110111", "101000" : "1010000101", "101001" : "1010011111",
    "101010" : "1010101100", "101011" : "1010110110", "101100" : "1011000010",
    "101101" : "1011011000", "101110" : "1011101011", "101111" : "1011110001",
    "110000" : "1100000110", "110001" : "1100011100", "110010" : "1100101111",
    "110011" : "1100110101", "110100" : "1101000001", "110101" : "1101011011",
    "110110" : "1101101000", "110111" : "1101110010", "111000" : "1110000000",
    "111001" : "1110011010", "111010" : "1110101001", "111011" : "1110110011",
    "111100" : "1111000111", "111101" : "1111011101", "111110" : "1111101110",
    "111111" : "1111110100"
}


codewords = [
    "0000000000", "0000011010", "0000101001", "0000110011", "0001000111",
    "0001011101", "0001101110", "0001110100", "0010000110", "0010011100",
    "0010101111", "0010110101", "0011000001", "0011011011", "0011101000",
```

```
    "0011110010", "0100000101", "0100011111", "0100101100", "0100110110",
    "0101000010", "0101011000", "0101101011", "0101110001", "0110000011",
    "0110011001", "0110101010", "0110110000", "0111000100", "0111011110",
    "0111101101", "0111110111", "1000000011", "1000011001", "1000101010",
    "1000110000", "1001000100", "1001011110", "1001101101", "1001110111",
    "1010000101", "1010011111", "1010101100", "1010110110", "1011000010",
    "1011011000", "1011101011", "1011110001", "1100000110", "1100011100",
    "1100101111", "1100110101", "1101000001", "1101011011", "1101101000",
    "1101110010", "1110000000", "1110011010", "1110101001", "1110110011",
    "1111000111", "1111011101", "1111101110", "1111110100"]
```

`distance()` berechnet die Hamming-Distanz (siehe Seite 27) zweier (gleich langer) Wörter aus Nullen und Einsen: Das ist die Anzahl der unterschiedlichen Stellen.

```
def distance(x,y):
    # Hamming-Distanz zweier Worte berechnen
    dist = 0
    for i in range(0,len(x)):
        if x[i] != y[i]:  dist+=1
    return dist
```

Die Funktion `cc_error_correction` hängt an jeden 6-Bit-Block einen 4-Bit-Block mit Paritätsbits an. Diese sind Linearkombinationen aus den 6 Bits mit den Paritätsvektoren (0,0,0,0,1,1), (0,1,1,1,0,0), (1,0,1,1,0,1), (1,1,0,1,1,0). Als Übersetzungstabelle wird das *Dictionary* `hamming` verwendet.

```
### cc_error_correction
def cc_error_correction (s):
    if ERROR_CORRECTION == 0:
        result = s       #  Fehlerkorrektur deaktiviert
    else:
        result = ""
        for i in range (0,len(s)/6):
            partword = s[6*i:6*i+6]
            result += hamming[partword]
    return result
### end cc_error_correction
```

Die folgende Funktion dekodiert ein 10-Bit-Wort. Dazu bestimmt sie die Hamming-Distanz zu allen gültigen (in `codewords` gespeicherten) Code-Wörtern; das Code-Wort, für das diese Distanz minimal ist, ist beim Auftreten maximal eines Bitfehlers minimal. Die ersten sechs Stellen des Code-Worts ergeben dann das ursprüngliche Wort.

```
def cc_error_decode_one (x):
    result = ""
    dist=50000  # große Länge als Anfangswert
    for w in codewords:
        if distance (x, w) < dist:
            result = w
            dist = distance (x, w)
    return result[0:6]
```

Die Funktion `cc_error_decode` macht die Fehlerkorrekturkodierung rückgängig, dazu wandelt sie jeweils 10-Bit-Blöcke mit Hilfe der oben stehenden Funktion `cc_error_decode_one` wieder in 6-Bit-Blöcke um.

Neben dem Eingabe-String wird ein Parameter `index` übergeben, über den festgelegt wird, dass ein Anfangsstück des Strings ignoriert wird – das ist nur für die Rückübersetzung unvollständiger 10-Bit-Blöcke notwendig und wird bei der Statusanzeige des Daemon-Prozesses verwendet.

```
### cc_error_decode
def cc_error_decoder (s,index):
    if ERROR_CORRECTION == 0:
        result = s       #  Fehlerkorrektur deaktiviert
    else:
        result = s[:index]
        t = s[index:]    #  Bits am Anfang ignorieren (unvollstaendig)
        for i in range (0,len(t)/10):
            partword = t[10*i:10*i+10]
            result += cc_error_decode_one(partword)
    return result
### end cc_error_decode
```

C.1.2 Konverter

Es folgen zwei Funktionen, die ASCII-Text in 6-Bit-Block-Sequenzen und zurück konvertieren.

`cc_ascii_to_bits` wandelt eine ASCII-Nachricht in eine Folge aus Nullen und Einsen. Es werden nur Zeichen zwischen den ASCII-Werten 32 und 95 kodiert – das entspricht einem Alphabet mit 64 Zeichen, so dass 6 Bit für die Kodierung benötigt werden. Kleinbuchstaben werden in Großbuchstaben gewandelt, und alle anderen Zeichen außerhalb des gültigen Bereichs werden zu Leerzeichen. Zusätzlich werden zur Kennzeichnung von Anfang und Ende eine Startsequenz (ASCII 95, 32; wird zu „111111000000“) und eine Endsequenz (ASCII 32, 95) ergänzt.

```
### cc_ascii_to_bits
def cc_ascii_to_bits (s):
    result = "111111000000"         # Start-Sequenz: 111111000000
    s = upper(s)
    for c in s:
        if c < chr(32) or c > chr (95): c = chr(32)
        v = ord(c)-32        # i jetzt zwischen 0 und 63
        for i in [5,4,3,2,1,0]:
            if v & (2**i):
                result=result+"1"
            else:
                result=result+"0"
    result = result+"000000111111"  # End-Sequenz: 000000111111
    return result
### end cc_ascii_to_bits
```

`cc_bits_to_ascii` konvertiert in die andere Richtung: Aus (0,1)-Folgen werden wieder ASCII-Zeichenketten, wobei je sechs Bits in ein ASCII-Zeichen (mit ASCII-Wert zwischen 32 und 95) umgewandelt werden.

Als zweites Argument wird die Anzahl der am Anfang zu ignorierenden Bits angegeben – diese Funktion ist für die Statusabfrage des Daemons notwendig, der mitzählt, wie viele Bits (eines kodierten Zeichens) bereits übertragen und danach

aus dem Speicher gelöscht wurden. So wird eine Fehlinterpretation „unvollständiger“ kodierter Zeichen am Anfang der (0,1)-Sequenz vermieden.

```
### cc_bits_to_ascii
def cc_bits_to_ascii (s,index):
    result = ""
    pos = 0
    v = 0
    if index != 0: result=result+"?"
    for c in s[index:]:
        if c == "1": v = v+2**(5-pos)
        pos = pos+1
        if pos == 6:
            pos = 0
            result = result + chr (v+32)   # Zeichen anhängen
            v = 0
    return result
### end cc_bits_to_ascii
```

C.1.3 Service-Funktionen

Der folgende Abschnitt enthält die Daemon-Funktionen, über die neue IP-Adressen eingetragen, nicht mehr verwendete Adressen gelöscht, Nachrichten in die FIFO-Liste zu einer IP-Adresse eingetragen und einzelne Nachrichtenbits versendet werden; auch die Statusabfrage fällt in diesen Bereich.

Die Funktion `cc_add_ip` trägt eine neue IP-Adresse als Kommunikationspartner für verdeckte Kommunikation ein. Dazu wird im *Dictionary* `comlist` ein neuer Eintrag erzeugt, der aus leeren Statusinformationen und einer leeren Nachricht besteht.

Der Statusblock (a, b, c, d, e) hat folgende Inhalte:

- a: Zahl der versendeten (ASCII-) Zeichen
- b: Länge der Gesamtnachricht; wird bei mehreren Nachrichten stetig erhöht und gibt einen Überblick über die Gesamtkommunikation seit Eintragen der IP-Adresse
- c: Zeitpunkt, zu dem die erste Nachricht in die zugehörige FIFO-Liste eingetragen wurde
- d: Zeit des letzten Zugriffs (durch den Apache-Server)
- e: Bit-Counter – zählt mit, wie viele Bits eines kodierten ASCII-Zeichens bereits vom Apache-Server abgerufen wurden

```
### cc_add_ip
### Neue IP-Adresse ins Dictionary aufnehmen
def cc_add_ip (ip_address):
    if comlist.has_key (ip_address):
        # IP-Adresse bereits vorhanden
        return false
    else:
        # Neue Adresse eintragen
```

```
        status = [ 0, 0, 0, 0, 0 ]
                   # bytes_sent,total_msg_length,
                   # start_transmission(time),last_access(time),
                   # bit_count
        message = ""
        comlist[ip_address] = [ status, message ]
        return true
### end cc_add_ip
```

Über die Funktion cc_del_ip wird eine IP-Adresse wieder aus dem *Dictionary* comlist entfernt – dabei werden die noch gespeicherten Informationen zur Auswertung zurückgegeben.

```
### cc_del_ip
### IP-Adresse aus dem Dictionary löschen
def cc_del_ip (ip_address):
    if comlist.has_key (ip_address):
        ret_arg = comlist[ip_address]
        del comlist[ip_address]
        return ret_arg
    else:
        return false
### end cc_del_ip
```

cc_stat_ip gibt den *Dictionary*-Eintrag zu einer IP-Adresse zurück, er wird von der Funktion cc_gen_status (weiter unten) in lesbarer Form aufbereitet.

```
### cc_stat_ip
### Status zu IP-Adresse zurückgeben
def cc_stat_ip (ip_address):
    if comlist.has_key (ip_address):
        return comlist[ip_address]
    else:
        return false
### end cc_stat_ip
```

Mit cc_add_msg wird eine neue Nachricht für einen Kommunikationspartner in die FIFO-Liste der angegebenen IP-Adresse eingetragen. Die Nachricht wird über cc_ascii_to_bits zunächst in Bitkodierung gewandelt und über cc_error_correction anschließend fehlerkorrekturkodiert, falls die Variable ERROR_CORRECTION entsprechend gesetzt ist. Statusinformationen im *Dictionary*-Eintrag der IP-Adresse werden angepasst.

```
### cc_add_msg (ip_address, message)
### Nachricht für IP-Adresse anhängen
def cc_add_msg (ip_address, new_message):
    if comlist.has_key (ip_address):
        [ status, message ] = comlist[ip_address]
        message = message + cc_error_correction( cc_ascii_to_bits( new_message ) )
              # Nachricht anhängen, ggfs. mit Fehlerkorrektur
        status[1] = status[1] + len(new_message) + 4       # Länge
        if status[2] == 0: status[2] = time.time()         # Start der Übertr.
        comlist[ip_address] = [ status, message ]
        return true
    else:
        return false
### end cc_add_msg
```

cc_add_msgfile hat die gleiche Aufgabe wie cc_add_msg, liest aber die Nachricht aus einer angegebenen Datei statt aus den Aufrufparametern.

```
### cc_add_msgfile (ip_address, messagefile)
### Nachricht für IP-Adresse aus Datei lesen und anhängen
def cc_add_msgfile (ip_address, messagefile):
    if comlist.has_key (ip_address):
        try:
            f = open (messagefile,"r")
            new_message = f.read()     # Nachricht aus Datei lesen
            f.close()
        except:
            return false

        [ status, message ] = comlist[ip_address]
        message = message +  cc_error_correction( cc_ascii_to_bits( new_message ) )
              # Nachricht anhängen, ggfs. mit Fehlerkorrektur
        status[1] = status[1] + len(new_message) + 4       # Länge
        if status[2] == 0: status[2] = time.time()         # Start der Übertr.
        comlist[ip_address] = [ status, message ]
        return true
    else:
        return false
### end cc_add_msgfile
```

cc_gen_status übernimmt die Aufbereitung der Statusinformationen in lesbarer Form. Der Rückgabe-String hat den Aufbau

```
Info for IP address 192.168.1.1
Chars sent:  0,0
Msg length:  0
Start Trans: Sun Jan  9 16:08:55 2005
Last access: Sun Jan  9 16:18:45 2005
Message: TEST-NACHRICHT
```

Aufrufparameter sind die IP-Adresse und der Inhalt des *Dictionary*-Eintrags zu dieser IP-Adresse (wie er von cc_stat_ip zurückgegeben wird).

```
### cc_gen_status
def cc_gen_status (ip,result):
    [ status, message ] = result
    ret_msg = "Info for IP address "+ip+'\n' + \
              "Chars sent:  "+str(status[0])+','+str(status[4])+'\n' + \
              "Msg length:  "+str(status[1])+'\n'
    date1 = status[2]
    date2 = status[3]
    if date1 == 0:
        date1_str = "--"
    else:
        date1_str = time.ctime(date1)
    if date2 == 0:
        date2_str = "--"
    else:
        date2_str = time.ctime(date2)

    # Index berechnen
    ind = (-status[4]) % CHARLENGTH
    ret_msg = ret_msg + \
              "Start Trans: "+date1_str+'\n' + \
```

```
              "Last access: "+date2_str+'\n' + \
              "Message:     "+cc_bits_to_ascii(
                                cc_error_decoder( message, ind ), ind )
    return ret_msg
### end cc_gen_status
```

Während alle bisherigen Service-Funktionen der Kommunikation zwischen Daemon und Kontrollprogramm dienen, ist `cc_send_bit` für die Abfrage durch den Apache-Server zuständig. Aufgerufen mit der IP-Adresse, gibt diese Funktion 0 oder 1 zurück, abhängig davon, ob das nächste Nachrichtenbit 0 oder 1 ist. Das vordere Bit in der FIFO wird dabei entfernt, und die Statusinformationen werden angepasst.

Insbesondere werden der Bitzähler und der Zähler der versendeten (ASCII-) Zeichen angepasst, wobei hier durch Vergleich mit der Variable `CHARLENGTH` (6 oder 10) berücksichtigt wird, ob Fehlerkorrektur eingesetzt wird oder nicht.

Im Fall einer unbekannten IP-Adresse wird 0 zurückgegeben: Das kennzeichnet eine unverzögerte Übertragung; bei IP-Adressen, die keine Kommunikationspartner sind, soll keine Verzögerung stattfinden.

```
### cc_send_bit
### Prüfen, ob Adresse ein Client ist, dann 1 Bit aus Nachricht
### entfernen
def cc_send_bit (ip_address):
    if comlist.has_key (ip_address):
        [ status, message ] = comlist[ip_address]
        # Ein Bit vorne abschneiden
        if message=="": return "0" # keine Nachricht gespeichert!
        msg_bit = message[0]   # Verzögern? "0" oder "1"
        message = message[1:]
        # Zähler anpassen
        bit_count = status[4]+1
        if bit_count == CHARLENGTH:
            status[0]+=1  # bytes_sent erhöhen
            status[1]-=1  # msg_length erniedrigen
            bit_count = 0
        status[4] = bit_count
        status[3] = time.time()  # letzten Zugriff anpassen
        # Daten zurück schreiben
        comlist[ip_address] = [ status, message ]
    else:
        msg_bit = "0"   # IP-Adresse unbekannt
    return msg_bit
### end cc_send_bit
```

C.1.4 Socket-Handler

Der folgende Code bearbeitet Anfragen an den Daemon. Jede Anfrage wird zunächst in ihre einzelnen Parameter zerlegt (die durch Leerzeichen voneinander getrennt sind). Es gibt maximal vier Parameter, der vierte (bei gültiger Syntax ist das die Nachricht oder der Name der Datei, welche die Nachricht enthält) darf auch weitere Leerzeichen enthalten.

```
### ccdaemon
### Der Handler fuer Anfragen
class ccdaemon(SocketServer.BaseRequestHandler):
    def handle (self):
        f = self.request.makefile()
        req = f.readline ()
        print "Request: '"+req+"'\n"
        # Anfrage von abschließenden Zeilenumbruchzeichen befreien
        while req[-1:] in ['\n','\r']:
            req = req[:-1]
        req_parts = req.split(" ",3)   # maximal vier Argumente
```

Anfragen beginnen stets mit „C“ oder „A“: Erstere kommen vom Kontrollprogramm, letztere vom Apache-Server.

```
        if req_parts[0] == "C":
            # Anfrage vom Kontrollprogramm
            if req_parts[1] == "add":
                # IP-Adresse hinzufügen
                result = cc_add_ip (req_parts[2])
                if result:
                    ret_msg = "OK"
                else:
                    ret_msg = "Error: IP address exists"
            elif req_parts[1] == "del":
                # IP-Adresse löschen
                ip = req_parts[2]
                result = cc_del_ip (ip)
                if result:
                    ret_msg = cc_gen_status (ip,result)+'\n'+"IP removed"
                else:
                    ret_msg = "Error: IP address "+ip+" not found"
            elif req_parts[1] == "msg":
                # Nachricht anhängen
                ip = req_parts[2]
                msg = req_parts[3]
                result = cc_add_msg(ip,msg)
                if result:
                    ret_msg = "OK"
                else:
                    ret_msg = "Error: IP address "+ip+" not found"
            elif req_parts[1] == "msgfile":
                # Nachricht aus Datei anhängen
                ip = req_parts[2]
                msgfile = req_parts[3]
                result = cc_add_msgfile(ip,msgfile)
                if result:
                    ret_msg = "OK"
                else:
                    ret_msg = "Error: IP address "+ip+" not found"
            elif req_parts[1] == "stat":
                # Status-Abfrage
                ip = req_parts[2]
                result = cc_stat_ip (ip)
                if not result:
                    ret_msg = "Error: IP address "+ip+" not found"
                else:
                    ret_msg = cc_gen_status (ip,result)
                # end Statusabfrage
            elif req_parts[1] == "allstat":
                ret_msg = ""
```

```
                for ip in comlist.keys():
                    result = cc_stat_ip (ip)
                    ret_msg = ret_msg + '\n'+ cc_gen_status (ip,result)
            else:
                ret_msg = "Error: Unknown Control argument"

        elif req_parts[0] == "A":
            # Anfrage von Apache
            ip = req_parts[1]
            ret_msg = cc_send_bit (ip)

        else:
            # Fehler
            ret_msg = "Error: Unknown command"

        self.request.send (ret_msg+"\n")
        print "Reply: "+ret_msg
        sys.stdout.flush()
        f.close ()
### end ccdaemon
```

C.1.5 Hauptprogramm

Die Hauptroutine des Daemons besteht nur aus zwei Zeilen: Es wird ein neuer TCP-Server erzeugt und an die obenstehende Handler-Klasse gebunden, danach wird der Server aktiviert.

```
### main()
### Server starten, Endlosschleife
server = SocketServer.TCPServer (("", CC_PORT), ccdaemon)
server.serve_forever()
### end main()
```

C.2 Der Proxy-Server ccproxy

Auf der Client-Seite empfängt der Proxy-Server `ccproxy` die Daten und wertet die Verzögerungen aus, die vom Apache-Server vor jedem 4-KByte-Block eingefügt oder ausgelassen wurden. Die Analyse der Verzögerungen findet erst später statt.

Die hier abgedruckte Version ist die Netzwerkversion, welche die Effekte der Segmentierung ausgleicht, indem erst nach dem Empfang von je 4096 Bytes die Verzögerung ausgewertet wird. (Im lokalen Fall kam es aufgrund der größeren MTU (*Maximum Transmission Unit*) nicht zu Fragmentierung, so dass dort eine simplere Version dieses Programmes eingesetzt wurde.)

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
#
# cc proxy
# Teil der Informatik-Diplomarbeit von Hans-Georg Eßer
# http://privat.hgesser.com/docs/Info-Diplom/
# (c) 2004, 2005 Hans-Georg Eßer
```

```
#
# proxy/ccproxy.py


### Netzwerkversion
### prüft, dass wirklich 4096 Bytes empfangen wurden, und schreibt erst
### dann die Zeit
```

In die Protokolldatei, die über die Variable LOGFILE definiert wird, schreibt der Proxy die gemessenen Verzögerungszeiten. Diese Protokolldatei wird später von ccanalyse ausgewertet.

```
# Globale Variablen

LOGFILE="/home/esser/diplom.local/myproxy/myproxy.log"


# Init
from socket import *
from time import sleep,time,ctime
from string import split
from sys import stdout
```

Die Funktion serve implementiert die vollständige Proxy-Funktionalität: Sie wertet Proxy-konforme HTTP-Anfragen aus, lädt die angeforderten Daten 4-KByte-blockweise vom Web-Server herunter, leitet sie weiter und misst die Verzögerungen der einzelnen Pakete.

```
# serve(): Routine, die mit einem Client spricht.
def serve(c,addr):
```

Zunächst wird die Anfrage ausgewertet:

```
    client_ip=addr[0]
    msg=c.recv(2048)
    lines = split(msg,'\n')
    for i in range(len(lines)):
        lines[i] = lines[i][:-1]
    getcmd = lines[0]
    getcmdparts = split(getcmd)     # GET-Zeile filtern
    url=getcmdparts[1]              # URL aus "GET http://... HTTP/1.x"
    [null1,null2,serverport,rest]=split(url,"/",3)  # (genau 3 Teile)
    if ":" in serverport:
        [server,port]=split(serverport,":")
        port=int(port)
    else:
        server=serverport; port=80
```

Jetzt enthalten die Variablen server, port und rest den zu kontaktierenden Server, den Port und den lokalen Teil der HTTP-URL.

Der Proxy baut dann eine Verbindung zum HTTP-Server auf und leitet die Anfrage an den Server weiter.

```
    # Web-Seite holen und Timing ueberwachen
    s_web = socket(AF_INET, SOCK_STREAM)
    s_web.connect((server,port))
```

```
    for l in lines[:-2]:     # alle Zeilen ausser den letzten 2
        s_web.send(l+"\n")
    s_web.send("\n")
```

Die folgende Schleife liest in 4-KByte-Blöcken (festgelegt durch die Variable psize) die Daten vom Web-Server und reicht sie an den Client weiter. In addedtime werden die (als Differenz von t1 und t0) gemessenen Übertragungszeiten aufaddiert, bis ein vollständiger 4-KByte-Block gelesen wurde. Diese aufsummierten Zeiten werden im Array times gespeichert.

s_web ist der Socket für die Kommunikation mit dem Apache-Server, c ist die Verbindung mit dem Client.

```
    # Blockweise holen und zustellen
    done=0
    psize=4096                      # Blockgröße für recv(): 4 KByte
    times=[]                        # Hier die Wartezeiten speichern
    addedtime = 0.0                 # Evtl. mehrere recv()-Aufrufe
    recvsize = 0                    # Auf 4096 hochzählen
    while not done:
        t0=time()
        result=s_web.recv(psize-recvsize)
        t1=time()
        recvsize += len(result)
        addedtime += (t1-t0)
        if recvsize >= psize:
            times.append(addedtime)   # Verzögerung speichern
            recvsize -= psize
            addedtime = 0.0
        c.send(result)
        if result=="":
            done=1
    s_web.close()
```

Nach der vollständigen Übertragung der Daten werden die Übertragungszeiten aus times in ein lesbares Format konvertiert und in die Protokolldatei geschrieben.

Hier erfolgt auch eine (vorläufige) Interpretation der Übertragungszeiten, die aber im späteren Analyseprozess nicht verwendet wird. Sie dient eher der Kontrolle.

```
    secret=""
    timestring=""
    for t in times:
        timestring = timestring + ( "%1.5f" % t ) + " "
        if t > 0.01:
            secret+="1"
        else:
            secret+="0"
    logtime = ctime()
    log.write ( logtime+": "+timestring[:-1]+'\n' )
    log.write ( logtime+": "+secret+'\n' )
# end serve()
```

Die Hauptroutine erzeugt einen neuen TCP-Socket, der an Port 21000 gebunden wird. Die Bearbeitung einzelner Anfragen erfolgt hier (anders als im Daemon-Programm, das die SocketServer-Klasse verwendet) manuell.

```
# main()
s = socket(AF_INET, SOCK_STREAM)   # TCP Socket erzeugen
s.bind(("",21000))                 # An Port 21000 binden
s.listen(1)                        # Eine Verbindung zulassen

while 1:
    client,addr = s.accept()       # Verbindung annehmen
    log = open(LOGFILE,"a")        # Logfile öffnen, Append-Mode
    serve(client,addr)             # Mit Client arbeiten
    log.close()                    # Logfile schließen
    client.close()                 # Verbindung beenden
```

C.3 Das Analyseprogramm ccanalyse

Der Proxy-Server schreibt nur die Verzögerungszeiten in eine Protokolldatei und verzichtet auf eine weitergehende Interpretation – diese Aufgabe übernimmt das Analyseprogramm `ccanalyse`. Es verwendet dabei die statistischen Methoden, die in Kapitel 3.6 vorgestellt wurden.

Im Fall aktivierter Fehlerkorrektur werden vor der Ausgabe der Ergebnisse 10-Bit-Blöcke in 6-Bit-Blöcke zurückgewandelt.

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
#
# cc analyser
# Teil der Informatik-Diplomarbeit von Hans-Georg Eßer
# http://privat.hgesser.com/docs/Info-Diplom/
# (c) 2004, 2005 Hans-Georg Eßer
#
# proxy/ccanalyse/ccanalyse.py
```

Über die Variable `ERROR_CORRECTION` wird festgelegt, ob die zu analysierenden Daten mit oder ohne Fehlerkorrektur übertragen wurden. Für aktivierte Fehlerkorrektur ist der Wert auf 1 zu setzen.

```
# Fehlerkorrektur an: 1, aus: 0
ERROR_CORRECTION=1

from math import sqrt
from re import sub
from string import upper,find
```

Die folgenden drei Funktionen berechnen die statistischen Funktionen, die in Kapitel 2.9 beschrieben wurden. Abweichend von der offiziellen Median-Definition wählt `median()` immer einen der in der Liste vorhandenen Werte aus und bildet bei gerader Elementzahl nicht den Mittelwert der beiden mittleren Werte. Für die Aufteilung eines Arrays in zwei „obere“ und „untere Hälften“ spielt das keine Rolle.

```
def mittelwert(seq):
   # Mittelwert eines Arrays zurückgeben
   m=0.0
   for i in seq:
      m = m+i
```

```
    return m/len(seq)

def abweichung(seq):
    # Standardabweichung eines Arrays zurückgeben
    mw = mittelwert(seq)
    n = len(seq)
    sum = 0
    for v in seq:
        sum = sum + (v-mw)**2
    return sqrt ( sum / (n-1) )

def median(seq):
    # Median-Wert eines Arrays zurückgeben
    seq2 = []
    for v in seq: seq2.append(v)
    seq2.sort()
    return seq2 [ len(seq2)/2 ]
```

Die Funktion `medsplit()` nimmt als Argumente eine Liste von Zahlen und einen Trenner. Sie gibt zwei Listen zurück, von denen die erste Liste alle Werte enthält, die kleiner als der Trenner sind; die zweite Liste enthält die übrigen Werte.

```
def medsplit(seq,med):
    s1 = []  # Werte < med
    s2 = []  # Werte >= med
    for v in seq:
        if v < med:
            s1.append(v)
        else:
            s2.append(v)
    return (s1,s2)
```

Über `fixvals()` wird die mit `medsplit()` aufgeteilte Liste noch weiter bearbeitet: Werte, die näher am Mittelpunkt der jeweils anderen Liste sind, werden zwischen den beiden Listen ausgetauscht.

```
def fixvals(A,B):
    a = mittelwert(A)
    b = mittelwert(B)
    A2B = []   # Werte, die von A nach B wandern sollen
    B2A = []   # Werte, die von B nach A wandern sollen

    for v in A+B:
        if v in A and abs(v-a) > abs (v-b): A2B.append(v)
        if v in B and abs(v-b) > abs (v-a): B2A.append(v)

    for v in A2B:
        A.remove(v)
        B.append(v)
    for v in B2A:
        B.remove(v)
        A.append(v)

    if max(A) > min(B): print "Fehler in fixvals()"

    return (A,B)
```

Die Liste `codewords` enthält (wie im Daemon-Programm, siehe oben) die 10 Bit langen Code-Worte.

```
codewords = [
    "0000000000", "0000011010", "0000101001", "0000110011",
    "0001000111", "0001011101", "0001101110", "0001110100",
    "0010000110", "0010011100", "0010101111", "0010110101",
    "0011000001", "0011011011", "0011101000", "0011110010",
    "0100000101", "0100011111", "0100101100", "0100110110",
    "0101000010", "0101011000", "0101101011", "0101110001",
    "0110000011", "0110011001", "0110101010", "0110110000",
    "0111000100", "0111011110", "0111101101", "0111110111",
    "1000000011", "1000011001", "1000101010", "1000110000",
    "1001000100", "1001011110", "1001101101", "1001110111",
    "1010000101", "1010011111", "1010101100", "1010110110",
    "1011000010", "1011011000", "1011101011", "1011110001",
    "1100000110", "1100011100", "1100101111", "1100110101",
    "1101000001", "1101011011", "1101101000", "1101110010",
    "1110000000", "1110011010", "1110101001", "1110110011",
    "1111000111", "1111011101", "1111101110", "1111110100"]
```

`distance()` berechnet die Hamming-Distanz zweier Listen, also die Anzahl der Positionen, an denen sich die beiden Listen unterscheiden.

```
### distance()
### Hamming-Distanz zwischen zwei Worten
def distance(x,y):
    dist = 0
    for i in range(0,len(x)):
        if x[i] != y[i]:  dist+=1
    return dist
### end distance()
```

Die folgenden zwei Funktionen dienen der Dekodierung der Fehlerkorrektur. Dazu wird das Codewort mit der geringsten Hamming-Distanz zur empfangenen 10-Bit-Sequenz gesucht. Vom Codewort werden dann die letzten 4 Bit abgeschnitten, um das Wort zu bestimmen, das mit größter Wahrscheinlichkeit gesendet wurde.

```
### cc_error_decode_one
### Diese Funktion wandelt einen fehlerkorrektur-kodierten
### 10-Bit-Block in einen 6-Bit-Block zurück.
def cc_error_decode_one (x):
    result = ""
    dist=50000  # große Länge als Anfangswert
    for w in codewords:
        if distance (x, w) < dist:
            result = w
            dist = distance (x, w)
    return result[0:6]
### end cc_error_decode_one


### cc_error_decode
### Diese Funktion wandelt fehlerkorrektur-kodierte 10-Bit-Blöcke
### wieder in 6-Bit-Blöcke um.
def cc_error_decoder (s,index):
    if ERROR_CORRECTION == 0:
        result = s       #  Fehlerkorrektur deaktiviert
    else:
        result = s[:index]
        t = s[index:]    #  Bits am Anfang ignorieren (unvollstaendig)
        for i in range (0,len(t)/10):
```

```
            partword = t[10*i:10*i+10]
            result += cc_error_decode_one(partword)
    return result
### end cc_error_decode
```

Die beiden Funktionen `cc_bits_to_ascii()` und `cc_ascii_to_bits()` funktionieren wie die gleichnamigen Funktionen in `ccdaemon.py` (siehe oben). Es gibt eine Ausnahme: `cc_bits_to_ascii()` fügt in diesem Programm keine Anfangs- und Endmarkierungen (111111000000, 000000111111) hinzu, da es nur zum Vergleich mit der empfangenen Nachricht verwendet wird, bei der die Markierungen bereits entfernt wurden.

```
### cc_bits_to_ascii
### Diese Funktion wandelt einen 01-String wieder in ASCII zurueck.
### als zweites Argument wird die Anzahl der am Anfang zu ignorierenden
### Bits ausgegeben.
def cc_bits_to_ascii (s,index):
   result = ""
   pos = 0
   v = 0
   if index != 0: result=result+"?"
   for c in s[index:]:
      if c == "1": v = v+2**(5-pos)
      pos = pos+1
      if pos == 6:
         pos = 0
         result = result + chr (v+32)   # Zeichen anhängen
         v = 0
   return result
### end cc_bits_to_ascii


### cc_ascii_to_bits
### Diese Funktion wandelt eine ASCII-Nachricht in eine Folge
### aus Nullen und Einsen. Es werden nur Zeichen zwischen ASCII 32 und 95
### kodiert (= 64 Zeichen = 6 Bit).
# modifizierte Version: ohne Header und Footer
def cc_ascii_to_bits (s):
   result = ""  # ohne (!) Start-Sequenz
   s = upper(s)
   for c in s:
      if c < chr(32) or c > chr (95): c = chr(32)
      v = ord(c)-32        # i jetzt zwischen 0 und 63
      for i in [5,4,3,2,1,0]:
         if v & (2**i):
            result=result+"1"
         else:
            result=result+"0"
   return result
### end cc_ascii_to_bits
```

Die Funktion `bitfehler()` misst die Abweichungen in zwei Strings. Sie unterscheidet sich nur geringfügig von `distance()`, indem sie Argumente ungleicher Länge akzeptiert.

```
### bitfehler
### Vergleich zweier {0,1}-Wörter; Rückgabe der Anzahl der Abweichungen
def bitfehler (s1,s2):
```

```
   length = min (len(s1),len(s2))
   s1 = s1[:length]  # abschneiden
   s2 = s2[:length]
   errcount = []
   for i in range(0,length):
      if s1[i] != s2[i]: errcount.append(i)
   return errcount
### end bitfehler
```

Hier beginnt das Hauptprogramm von `ccanalyse.py`. Es liest die Eingabedatei ein, stellt verschiedene Berechnungen an und gibt die gefundenen Werte aus.

```
infile=open("data.in","r")
s=infile.read()                      # Einlesen der Datei
infile.close()

vals=[]
for v in s.split():
   vals.append(float(v))             # Strings in Zahlen umwandeln

med = median(vals)                   # Median und
mw = mittelwert(vals)                # Mittelwert berechnen
(vals1,vals2) = medsplit(vals,med)   # Liste splitten

print "Gesamtwerte"                  # Werte vor Aufruf von fixvals()
print "Median:     %1.10f" % med
print "Mittelwert: %1.10f" % mw
print "Abweichung: %1.10f" % abweichung(vals)

print "Unterer Block"
print "Minimum:    %1.10f" % min(vals1)
print "Maximum:    %1.10f" % max(vals1)
print "Mittelwert: %1.10f" % mittelwert(vals1)
print "Abweichung: %1.10f" % abweichung(vals1)

print "Oberer Block"
print "Minimum:    %1.10f" % min(vals2)
print "Maximum:    %1.10f" % max(vals2)
print "Mittelwert: %1.10f" % mittelwert(vals2)
print "Abweichung: %1.10f" % abweichung(vals2)

(vals1,vals2) = fixvals(vals1,vals2)

print "AUFRAEUMEN"                   # Werte nach Aufruf von fixvals()
print "Unterer Block"
print "Minimum:    %1.10f" % min(vals1)
print "Maximum:    %1.10f" % max(vals1)
print "Mittelwert: %1.10f" % mittelwert(vals1)
print "Abweichung: %1.10f" % abweichung(vals1)

print "Oberer Block"
print "Minimum:    %1.10f" % min(vals2)
print "Maximum:    %1.10f" % max(vals2)
print "Mittelwert: %1.10f" % mittelwert(vals2)
print "Abweichung: %1.10f" % abweichung(vals2)
```

Als Unterscheidungskriterium, welche Übertragungszeiten als 0 und welche als 1 interpretiert werden sollen, wird ein Trenner bestimmt. Die folgende Codezeile legt ihn auf $\max(\texttt{vals1}) + 0,001$ fest.

```
trenner = max(vals1)+0.0001
```

Danach wird anhand dieser Unterscheidung die ursprüngliche Nachricht rekonstruiert und in `msgbin` gespeichert.

```
msgbin = ""
for v in vals:
   if v < trenner:
      msgbin = msgbin+"0"
   else:
      msgbin = msgbin+"1"

print "Nachricht:␣",msgbin
```

Für den Fall, dass mit Fehlerkorrektur gearbeitet wurde, wird jetzt noch die Funktion `cc_error_decoder()` aufgerufen.

```
# Fehlerkorrektur anwenden, falls Variable gesetzt ist.
# Dadurch werden die 10-Bit-Blöcke wieder zu 6-Bit-Blöcken.
if ERROR_CORRECTION == 1:
   msgbin = cc_error_decoder (msgbin, 0)
```

Schließlich wird die Nachricht im Binär- und ASCII-Format ausgegeben; zum Vergleich folgt die Originalnachricht.

```
msgbin=msgbin[12:]
print "Nachricht: ",msgbin
print "Text:      ", cc_bits_to_ascii(msgbin,0)

originaltext = """O@ .,NN'.:9I1]D4<S';]&S>ASUL,K?L&,KO5'\SO<TOY"W5SGR6]D*;"""
             + """P+9&I@&AO2 =UA*VH>!]R%M.GP8Q)P@..ZO-SMU:$HT/Z7/!E$ *Y%.3"""
             + """HM^2#**5QS"*21(N _"""

print "Original:  ", originaltext
```

Zum Abschluss berechnet das Programm die absolute und prozentuale Fehlerzahl.

```
bf = bitfehler (msgbin,cc_ascii_to_bits(originaltext))
print "Bitfehler: ", len(bf),": ",bf
print "Laenge:    ", len(cc_ascii_to_bits(originaltext))
print "Fehler %:  ", (len(bf)+0.0)/(len(cc_ascii_to_bits(originaltext))+0.0)*100
```

C.4 Die Apache-Funktion `ap_covert_check_ip`

Der Apache-Server verzögert in den Funktionen `ap_send_fd_length()` und `ap_send_mmap()` gegebenenfalls den Versand einzelner 4-KByte-Blöcke. Um herauszufinden, ob ein Paket verzögert werden soll, wird aus diesen Funktionen heraus die neu geschriebene Funktion `ap_covert_check_ip()` aufgerufen, der als Argument die IP-Adresse übergeben wird.

`ap_covert_check_ip()` baut eine Verbindung zum Daemon-Prozess auf, übermittelt ihm die IP-Adresse und erhält als Antwort „0“ oder „1“ zurück. Der Rückgabewert `reply` der Funktion ist – abhängig von der Antwort des Daemons – 0 oder 1.

```
int ap_covert_check_ip (char *ip) {
  /* Code fuer Socket-Aufbau von http://pont.net/socket/prog/tcpClient.c
     uebernommen und angepasst */
  int sd, rc, i;
  struct sockaddr_in localAddr, servAddr;
  struct hostent *h;
  static char daemon_request[30];
  static char daemon_reply[10];
```

Die ersten Zeilen im Programm bauen eine Verbindung zum Daemon auf:

```
  h = gethostbyname (AP_CC_HOSTNAME);
  /* Hostname AP_CC_HOSTNAME in include/covert_channel.h definiert */

  servAddr.sin_family = h->h_addrtype;
  memcpy((char *) &servAddr.sin_addr.s_addr, h->h_addr_list[0], h->h_length);
  servAddr.sin_port = htons(AP_CC_PORT);
  /* Port AP_CC_PORT in include/covert_channel.h definiert */

  /* create socket */
  sd = socket(AF_INET, SOCK_STREAM, 0);

  /* bind any port number */
  localAddr.sin_family = AF_INET;
  localAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  localAddr.sin_port = htons(0);

  bind(sd, (struct sockaddr *) &localAddr, sizeof(localAddr));

  /* connect to server */
  connect(sd, (struct sockaddr *) &servAddr, sizeof(servAddr));
```

Dann wird die Nachricht in der Form `A` *`IP-Adresse`* zusammengesetzt und an den Daemon geschickt.

```
  /* "A " + IP-Adresse + Newline an Daemon senden */
  strcpy (daemon_request, "A␣");
  strcat (daemon_request, ip);
  strcat (daemon_request, "\n");
  send (sd, daemon_request, strlen(daemon_request)+1, 0);
```

Der sendet eine Antwort, die von der Funktion interpretiert wird und aus der sie den Rückgabewert bestimmt.

```
  /* Antwort empfangen und auswerten, Verbindung schliessen */
  const int MAX_LEN=2;
  recv (sd, daemon_reply, MAX_LEN, 0);
  close (sd);
  int reply;
  if (daemon_reply[0] == '1')
    { reply = 1; }   /* nur 1, wenn "1\n" empfangen wurde */
  else
    { reply = 0; }

  return reply;
}
```

C.5 Berechnung von Hamming-Codes

Die Fehlerkorrektur verwendet einen Hamming-Code, der jede 6-Bit-Folge um vier Prüfsummenbits ergänzt. Der Code wurde mit dem hier abgedruckten Python-Programm gefunden, das systematisch alle möglichen Codes aufzählt und deren Hamming-Distanz bestimmt. Um die Korrektur eines Bitfehlers zu ermöglichen, musste ein Code mit Hamming-Distanz 3 gefunden werden.

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-

SRCLEN=6  # Anzahl der Bits in den zu kodierenden Worten
```

`create_words()` erzeugt eine Liste aller aus 0 und 1 zusammensetzbaren Worte der als Argument übergebenen Länge.

```
def create_words(len):
    # Erzeuge eine Liste aller Worte mit Länge len
    array = [[]]
    for i in range(0,len):
        newarray = []
        for v in array:
            for j in [0,1]:
                newarray.append (v+[j])
        array = newarray
    return array
```

Für eine Liste von Paritäts-Checks wendet `apply_parity_checks` alle Checks auf ein Wort an, fügt also die Prüfsummenbits hinzu.

```
def apply_parity_checks (word, checks):
    # Parity Checks (Summe: 0) anwenden
    newword = []  # Kopieren mit "newword = word" geht nicht...
    for i in range(0,len(word)): newword.append(word[i])

    for c in checks:
        parity = 0
        for i in range(0,len(c)):
            if c[i] == 1:   parity += word[i]
            if parity == 2: parity = 0
        newword.append(parity)
    return newword
```

`parity_code()` erzeugt einen Code, indem es auf eine Liste von Worten mit `apply_parity_checks()` eine Liste von Paritäts-Checks anwendet.

```
def parity_code(input,checks):
    # Aus einem Array von Eingabeworten und einer Liste
    # von Parity-Checks einen Code berechnen
    out = []
    for w in input:
        res = apply_parity_checks (w, checks)
        out.append(res)
    return out
```

`distance()` ist wieder die bereits bekannte Hamming-Distanz $h(x, y)$ zwischen zwei Worten x und y.

```
def distance(x,y):
    # Hamming-Distanz zweier Worte berechnen
    dist = 0
    for i in range(0,len(x)):
        if x[i] != y[i]:  dist+=1
    return dist
```

Die folgende Funktion berechnet die Hamming-Distanz eines Codes, also $\min\{h(a,b) \mid a,b \in \text{Code}\}$.

```
def hamming_distance(code):
    # Hamming-Distanz eines Codes berechnen (minimaler Abstand
    # distance(x,y) für alle x != y
    dist=50000  # große Länge als Anfangswert
    for x in code:
        for y in code:
            if x != y: dist = min (dist, distance(x,y))
    return dist
```

Der Rest der Funktion konstruiert in einer vierfachen Schleife alle möglichen Codes und berechnet deren Hamming-Distanz. Wird ein Code mit Distanz 3 oder größer gefunden, gibt das Programm ihn aus.

```
inputs=create_words(SRCLEN)

num = len(inputs)
for i1 in range(0,num):
    c1 = inputs[i1]
    for i2 in range(i1+1,num):
        c2 = inputs[i2]
        for i3 in range(i2+1,num):
            c3 = inputs[i3]
            for i4 in range(i3+1,num):
                c4 = inputs[i4]
                code = parity_code (inputs,[c1,c2,c3,c4])
                d = hamming_distance(code)
                if d > 2:
                    print "Hamming-Distanz␣Inputs:",d, "für␣Code", c1, c2, c3, c4
```

Anhang D

CD-Inhalt

Dieser Anhang beschreibt, welche Dateien sich auf der mitgelieferten CD befinden. Die CD besitzt ein hybrides Dateisystem (mit Rockridge-Erweiterungen für Linux-/ Unix-Rechner und Joliet-Erweiterungen für Windows-Rechner) und sollte sich auf allen aktuellen Plattformen lesen lassen.

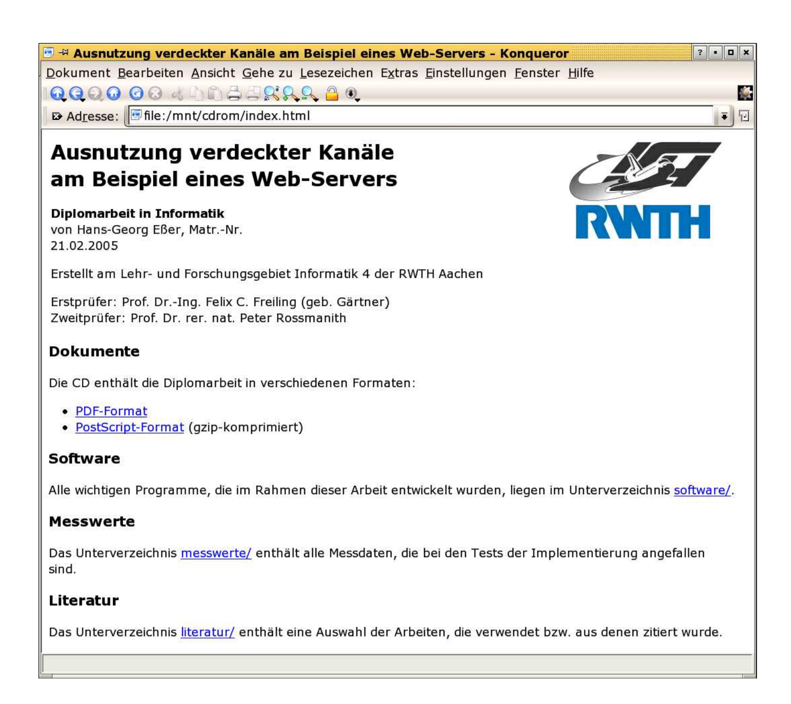


Abbildung D.1: Indexseite der beiliegenden CD.

Unter Linux kann die CD mit dem Befehl

```
mount /dev/cdrom /mnt/cdrom
```

eingebunden werden, wobei Gerätename und Mount-Punkt gegebenenfalls anzupassen sind.

Sollte die CD nicht lesbar sein, stellt der Autor der Arbeit auf Anfrage an `h.g.esser@gmx.de` ein ISO-Image der CD bereit.

D.1 Dokumente

Das Verzeichnis `docs/` enthält PostScript- und PDF-Versionen dieser Arbeit. Das PDF-Format kann unter allen aktuellen Betriebssystemen gelesen und ausgedruckt werden (z. B. mit dem Acrobat Reader; unter Linux auch mit `xpdf` oder `gv`). Die PostScript-Version kann mit Ghostscript (unter Linux zum Beispiel mit dem Ghostscript-Frontend `gv`) geöffnet werden.

D.2 Software

Im Verzeichnis `software` liegen alle im Rahmen dieser Arbeit implementierten Programme, im Einzelnen:

- Original-Apache-Quellen (Version 1.3.31)
- Patch der Apache-Quellen
- Gepatchte Apache-Quellen
- `ccdaemon`: Daemon, der auf dem Server läuft
- `ccproxy`: HTTP-Proxy-Server, der auf dem Client eingesetzt wird
- `ccctrl`: Kontrollprogramm (das via `netcat` den Daemon steuert)
- Analyse-Programme

D.3 Messdaten

Im Rahmen der Tests der Implementierung sind umfangreiche Messdaten angefallen. Alle erfassten Daten liegen auf der CD im Verzeichnis `messwerte`.

D.4 Literatur

Die im Literaturverzeichnis genannten Arbeiten wurden, soweit sie digital verfügbar waren, ebenfalls auf die CD aufgenommen. Sie liegen im Verzeichnis `literatur/`. Über den Link in dieses Verzeichnis ist ein Index der vorhandenen Dokumente verfügbar.

Vor einer eventuellen Vervielfältigung und Weitergabe der CD sollten diese Dateien aus urheberrechtlichen Gründen entfernt werden. Zwar sind alle Texte frei im Internet verfügbar, aber es handelt sich nicht um „freie“ Dokumentation.

Anhang E

Überwachung eines Dateitransfers

Um das Zusammenspiel von Apache-Server, TCP- und IP-Schicht besser zu verstehen, wurde ein beispielhafter Abruf der Testdatei (mit Verzögerungen) mit den Hilfsprogrammen `strace` und `ethereal` untersucht.

Für den Test wurde eine Übertragung der 5-MByte-Datei mit Verzögerung 0,12 s und ohne Fehlerkorrektur durchgeführt.

Die Protokolldateien `strace.log` und `ethereal.capture` liegen auf der Begleit-CD im Verzeichnis `messwerte/strace-ethereal/`, so dass die Analyse anhand dieser Daten nachvollziehbar ist. Die verwendete `ethereal`-Version ist 0.9.14, die Capture-Datei sollte sich aber auch mit anderen Versionen öffnen lassen. Für das reine Anzeigen der Datei sind keine Administratorrechte notwendig. Zusätzlich enthält `ethereal.txt` im gleichen Verzeichnis die über *File / Print* mit den Optionen *Print to File*, *Print detail* und *Expand all levels* erzeugte ASCII-Version.

E.1 Beobachtung von Apache mit `strace`

Um zu prüfen, ob Apache die Anweisung befolgt, 4 KByte große Blöcke zu senden, wird der Daemon mit

```
strace -f -o /tmp/strace.txt ./httpd -f ~esser/httpd.conf
```

gestartet. (Eine Einschränkung auf die wesentlichen Funktionsaufrufe ist mit dem Parameter `-e trace=write,read,open,connect,send,recv,nanosleep,writev` möglich, für den Test wurde aber ein volles Protokoll aufgezeichnet.) Die damit erfassten externen Aufrufe des Apache-Servers werden im Folgenden analysiert.

An den folgenden Zeilen aus einem `strace`-Mitschnitt kann man erkennen, dass der Apache-Server stets 4-KByte-Blöcke verschickt. Die ersten 19 Zeichen der bitkodierten Testnachricht sind

```
1111110000000100001
```

Damit sollten die Blöcke 1–6, 14 und 19 verzögert und die übrigen Blöcke (7–13, 15–18) unverzögert übertragen werden.

```
11489 fork()                              = 11490
11490 read(3, "GET␣http://hgesser.com:8080/rand"..., 4096) = 49
11490 open("/home/esser/diplom/apache-root/random.data", O_RDONLY) = 4
```

Die ersten drei Zeilen zeigen, wie ein neuer Apache-Prozess mit `fork()` erzeugt wird, der für die Bearbeitung der Anfrage zuständig ist. Er liest die HTTP-Anfrage des Clients und öffnet die angeforderte Datei.

Es folgen die Aufrufe, in denen der Prozess die ersten 4 KByte der Trägerdatei liest, beim Daemon nachfragt, ob das Paket verzögert werden soll, und schließlich nach der Verzögerung mit `nanosleep()` den HTTP-Header und den ersten 4-KByte-Block verschickt.

```
11490 read(4, "yI?\221d1i\377\261\356\313\376\270\215\2024TQ]0\237\37"..., 4096) = 4096
11490 connect(5, {sin_family=AF_INET, sin_port=htons(4999), sin_addr=inet_addr("217.160.179.171")}}, 16) = 0
11490 send(5, "A␣80.128.12.237\n\0", 17, 0) = 17
11490 recv(5, "1\n", 2, 0)              = 2
11490 nanosleep({0, 119999999}, {0, 119999999}) = 0
11490 writev(3, [{"HTTP/1.1␣200␣OK\r\nDate:␣Sun,␣20␣F"..., 256}, \
      {"yI?\221d1i\377\261\356\313\376\270\215\2024TQ]0\237\37"..., 4096}], 2) = 4352
```

Nur im ersten Block wurde `writev()` verwendet: So konnte Apache den Header und den ersten Block über einen Funktionsaufruf verschicken, ohne die beiden vorher mit `strcat()` zusammenzufügen.

Ab dem zweiten Durchgang werden 4-KByte-Blöcke mit `write()` geschrieben. Für den zweiten Block sieht das wie folgt aus:

```
11490 read(4, "F\305\206\17\363P\220\217\213\10\371\251(\336a]\252F{W"..., 4096) = 4096
11490 connect(5, {sin_family=AF_INET, sin_port=htons(4999), sin_addr=inet_addr("217.160.179.171")}}, 16) = 0
11490 send(5, "A␣80.128.12.237\n\0", 17, 0) = 17
11490 recv(5, "1\n", 2, 0)              = 2
11490 nanosleep({0, 119999999}, {0, 119999999}) = 0
11490 write(3, "F\305\206\17\363P\220\217\213\10\371\251(\336a]\252F{W"..., 4096) = 4096
```

Die Blöcke 3–6 werden auf gleiche Weise verschickt: Die Anfrage an den Daemon ergibt jeweils 1, so dass die Verzögerung ausgelöst wird. Im Protokoll ändert sich nur der Datenanteil, hier zum Beispiel für Block 3:

```
11490 read(4, "\211\253\361\3779\37\225\236\277(0\212\357\351\223sj\247"..., 4096) = 4096
11490 connect(5, {sin_family=AF_INET, sin_port=htons(4999), sin_addr=inet_addr("217.160.179.171")}}, 16) = 0
11490 send(5, "A␣80.128.12.237\n\0", 17, 0) = 17
11490 recv(5, "1\n", 2, 0)              = 2
11490 nanosleep({0, 119999999}, {0, 119999999}) = 0
11490 write(3, "\211\253\361\3779\37\225\236\277(0\212\357\351\223sj\247"..., 4096) = 4096
```

Auf den Abdruck der gleich aufgebauten Blöcke 4–6 wird verzichtet; weiter geht es mit Block 7 – das ist der erste Block, der ohne Verzögerung gesendet wird. Entsprechend erhält Apache bei der Daemon-Abfrage diesmal den Wert 0, so dass `nanosleep()` nicht aufgerufen wird.

```
11490 read(4, "M\342\\\274_\26\241}\2775\7H\vhR\303\307G\365\202F\217"..., 4096) = 4096
11490 connect(5, {sin_family=AF_INET, sin_port=htons(4999), sin_addr=inet_addr("217.160.179.171")}}, 16) = 0
11490 send(5, "A␣80.128.12.237\n\0", 17, 0) = 17
11490 recv(5, "0\n", 2, 0)              = 2
11490 write(3, "M\342\\\274_\26\241}\2775\7H\vhR\303\307G\365\202F\217"..., 4096) = 4096
```

Die Protokolldatei wurde auf abweichende Blocklängen durchsucht:

```
grep "write(3," strace.log | grep -v 4096
```

Es wurden aber keine gefunden. Apache sendet in jedem `write()`-Aufruf einen 4 KByte großen Datenblock. Aufrufe von `nanosleep()` finden an den vorgesehenen Stellen statt.

Damit ist für dieses Beispiel nachgewiesen, dass es nicht bereits auf HTTP-Ebene zur Reduzierung der Paketgröße kommt. Es bleiben als weitere Möglichkeiten TCP und IP.

E.2 Analyse mit `ethereal`

Die im vorherigen Abschnitt mit `strace` analysierte Datenübertragung wurde auf dem Client mit `ethereal` mitgeschnitten. Wenn im Folgenden Frame-Informationen ausgegeben werden, so enthalten diese immer nur die für das Verständnis wichtigen Informationen.

Frames 11–13: *Three way handshake*: In den Frames 11 bis 13 beginnt der Kontakt, indem zunächst die MSS ausgehandelt wird.

```
Frame 11 (74 bytes on wire, 74 bytes captured)
Internet Protocol, Src Addr: 192.168.1.1 (192.168.1.1), Dst Addr: 217.160.179.171 (217.160.179.171)
Transmission Control Protocol, Src Port: 54187 (54187), Dst Port: http-alt (8080), Seq: 3097798988,\
  Ack: 0, Len: 0
    Flags: 0x0002 (SYN)
    Options: (20 bytes)
        Maximum segment size: 1460 bytes
        Time stamp: tsval 456915685, tsecr 0

Frame 12 (74 bytes on wire, 74 bytes captured)
Internet Protocol, Src Addr: 217.160.179.171 (217.160.179.171), Dst Addr: 192.168.1.1 (192.168.1.1)
Transmission Control Protocol, Src Port: http-alt (8080), Dst Port: 54187 (54187), Seq: 960315118,\
  Ack: 3097798989, Len: 0
    Flags: 0x0012 (SYN, ACK)
    Options: (20 bytes)
        Maximum segment size: 1400 bytes
        Time stamp: tsval 1078560333, tsecr 456915685

Frame 13 (66 bytes on wire, 66 bytes captured)
Internet Protocol, Src Addr: 192.168.1.1 (192.168.1.1), Dst Addr: 217.160.179.171 (217.160.179.171)
Transmission Control Protocol, Src Port: 54187 (54187), Dst Port: http-alt (8080), Seq: 3097798989,\
  Ack: 960315119, Len: 0
    Flags: 0x0010 (ACK)
    Options: (12 bytes)
        Time stamp: tsval 456915692, tsecr 1078560333
```

Wie in Kapitel 2.8.1 dargestellt wurde, wird die MSS auf 1400 Byte festgelegt, was an der Verwendung eines DSL-Routers liegt. Tatsächlich schlagen die beteiligten Rechner die Standard-MSS von 1460 Byte (MTU 1500 – 40) vor, der Router ändert diesen Wert aber beim Übertragen der Pakete in beiden Richtungen auf 1400.

In den Frames 14 und 16 empfängt der Apache-Server die HTTP-Anfrage:

```
Frame 14 (115 bytes on wire, 115 bytes captured)
Internet Protocol, Src Addr: 192.168.1.1 (192.168.1.1), Dst Addr: 217.160.179.171 (217.160.179.171)
Transmission Control Protocol, Src Port: 54187 (54187), Dst Port: http-alt (8080), Seq: 3097798989,\
  Ack: 960315119, Len: 49
    Options: (12 bytes)
        Time stamp: tsval 456915692, tsecr 1078560333
Hypertext Transfer Protocol
    GET http://hgesser.com:8080/random.data HTTP/1.0\n
        Request Method: GET
```

```
Frame 16 (125 bytes on wire, 125 bytes captured)
Internet Protocol, Src Addr: 192.168.1.1 (192.168.1.1), Dst Addr: 217.160.179.171 (217.160.179.171)
Transmission Control Protocol, Src Port: 54187 (54187), Dst Port: http-alt (8080), Seq: 3097799038,\
  Ack: 960315119, Len: 59
    Options: (12 bytes)
        Time stamp: tsval 456915699, tsecr 1078560340
Hypertext Transfer Protocol
    User-Agent: Wget/1.8.2\n
    Host: hgesser.com:8080\n
    Accept: */*\n
    \n
```

Mit Frame 18 beginnt die Übertragung vom Apache-Server zum Client: Der Server schickt der HTTP-Header und den ersten 4 KByte großen Datenblock. Diese Übertragung wird vom TCP-Stack aufgeteilt, so dass insgesamt vier Frames für Header und ersten Block benötigt werden:

```
Frame 18 (1454 bytes on wire, 1454 bytes captured)
Internet Protocol, Src Addr: 217.160.179.171 (217.160.179.171), Dst Addr: 192.168.1.1 (192.168.1.1)
    Flags: 0x04
        .1.. = Don't fragment: Set
        ..0. = More fragments: Not set
Transmission Control Protocol, Src Port: http-alt (8080), Dst Port: 54187 (54187), Seq: 960315119,\
  Ack: 3097799097, Len: 1388
    Flags: 0x0010 (ACK)
    Options: (12 bytes)
        Time stamp: tsval 1078560361, tsecr 456915699
Hypertext Transfer Protocol
    HTTP/1.1 200 OK\r\n
    Date: Sun, 20 Feb 2005 11:14:19 GMT\r\n
    Server: Apache/1.3.31 (Unix)\r\n
    Last-Modified: Fri, 26 Nov 2004 20:52:34 GMT\r\n
    ETag: "18808e-500000-41a79792"\r\n
    Accept-Ranges: bytes\r\n
    Content-Length: 5242880\r\n
    Connection: close\r\n
    Content-Type: text/plain\r\n
    \r\n
    Data (1132 bytes)

Frame 20 (1454 bytes on wire, 1454 bytes captured)
Internet Protocol, Src Addr: 217.160.179.171 (217.160.179.171), Dst Addr: 192.168.1.1 (192.168.1.1)
    Flags: 0x04
        .1.. = Don't fragment: Set
        ..0. = More fragments: Not set
Transmission Control Protocol, Src Port: http-alt (8080), Dst Port: 54187 (54187), Seq: 960316507,\
  Ack: 3097799097, Len: 1388
    Flags: 0x0010 (ACK)
    Options: (12 bytes)
        Time stamp: tsval 1078560361, tsecr 456915699
Hypertext Transfer Protocol
    Data (1388 bytes)

Frame 22 (1454 bytes on wire, 1454 bytes captured)
Internet Protocol, Src Addr: 217.160.179.171 (217.160.179.171), Dst Addr: 192.168.1.1 (192.168.1.1)
    Flags: 0x04
        .1.. = Don't fragment: Set
        ..0. = More fragments: Not set
Transmission Control Protocol, Src Port: http-alt (8080), Dst Port: 54187 (54187), Seq: 960317895,\
  Ack: 3097799097, Len: 1388
    Flags: 0x0018 (PSH, ACK)
    Options: (12 bytes)
        Time stamp: tsval 1078560368, tsecr 456915720
Hypertext Transfer Protocol
    Data (1388 bytes)
```

```
Frame 24 (254 bytes on wire, 254 bytes captured)
Internet Protocol, Src Addr: 217.160.179.171 (217.160.179.171), Dst Addr: 192.168.1.1 (192.168.1.1)
    Flags: 0x04
        .1.. = Don't fragment: Set
        ..0. = More fragments: Not set
Transmission Control Protocol, Src Port: http-alt (8080), Dst Port: 54187 (54187), Seq: 960319283,\
  Ack: 3097799097, Len: 188
    Flags: 0x0018 (PSH, ACK)
    Options: (12 bytes)
        Time stamp: tsval 1078560368, tsecr 456915720
Hypertext Transfer Protocol
    Data (188 bytes)
```

Die HTTP-Datenblöcke dieser vier Frames enthalten 1132, 1388, 1388 und 188 Byte – in der Summe also 4096 Byte und damit den ersten von Apache verschickten Block.

In keinem der Pakete ist auf IP-Ebene das Flag `More Fragments` gesetzt, und alle Pakete haben das DF-Flag `Don't fragment` gesetzt. Zusammen mit den Informationen aus dem ersten Abschnitt ist damit klar, dass es auf TCP-Ebene zur Segmentierung kommt: Der Apache-Server verringert die Paketgröße nicht, und auf IP-Ebene ist keine Fragmentierung notwendig, da die Pakete bereits von TCP ausreichend segmentiert wurden.

Auf Client-Seite wurden in diesem Zeitraum fünf Blöcke mit den Längen 1388, 1388, 1320, 68 und 188 Byte gelesen – die Aufteilung des letzten 1388er-Blocks in zwei Einheiten zu 1320 und 68 Byte liegt daran, dass dieses 1388er-Paket mit dem PSH-Flag verschickt wurde – laut RFC 1122 [Bra89], Abschnitt 4.2.2.2, ist das vorgeschrieben:

> At the receiver, the PSH bit forces buffered data to be delivered to the application (even if less than a full buffer has been received).

Leider setzt sich die oben dargestellte Übertragungsart, in der jedes 4-KByte-Paket separat in (gegebenenfalls kleinere) TCP-Pakete verpackt wird, nicht fort.

Frames (28, 30, 32) haben HTTP-Datenlängen 1388, 1388 und 1320 (Summe wieder 4096), die Frames (34, 36, 38), (40, 42, 44), (46, 48, 50) und (52, 54, 56) ebenfalls.

Damit wurden die ersten sechs 4-KByte-Blöcke auf je drei Frames aufgeteilt. Ab Frame 58 (also ab dem siebten 4-KByte-Block) hört diese Aufteilung aber auf. Dem entspricht, dass die ersten sechs Blöcke verzögert und die folgenden Blöcke 7–13 unverzögert gesendet werden. Da es zwischen diesen Blöcken keine Wartezeiten gibt, fasst der TCP-Stack die enthaltenen Daten in maximal großen TCP-Paketen zusammen: Die Frames 58, 60, 62, 64, 65, 67, 68, 70, 71, 73, 74, 76, 77, 79, 80, 82 und 83 haben alle HTTP-Länge 1388, erst Frame 85 überträgt ein kleineres Paket mit HTTP-Länge 980. Die Gesamtlänge der in diesen Paketen übertragenen HTTP-Daten ist $17 \times 1388 + 980 = 24576 = 6 \times 4096$: Hier handelt es sich also um die folgenden sechs unverzögerten Pakete. Das nächste, unverzögerte Paket Nummer 13 wird in drei Segmente unterteilt; Paket 14 wird verzögert. Die Frames 87, 88 und 90 haben die HTTP-Längen 1388, 1388 und 1320 (Summe 4096).

In Frame 90 ist das PSH-Flag gesetzt, weil keine Daten nachkommen:

```
Frame 90 (1386 bytes on wire, 1386 bytes captured)
Internet Protocol, Src Addr: 217.160.179.171 (217.160.179.171), Dst Addr: 192.168.1.1 (192.168.1.1)
Transmission Control Protocol, Src Port: http-alt (8080), Dst Port: 54187 (54187), Seq: 960367303,\
  Ack: 3097799097, Len: 1320
    Flags: 0x0018 (PSH, ACK)
    Options: (12 bytes)
        Time stamp: tsval 1078560447, tsecr 456915800
Hypertext Transfer Protocol
    Data (1320 bytes)
```

RFC 1122 [Bra89] sagt außerdem zum PSH-Flag:

> When an application issues a series of SEND calls without setting the PUSH flag, the TCP MAY aggregate the data internally without sending it. Similarly, when a series of segments is received without the PSH bit, a TCP **MAY** queue the data internally without passing it to the receiving application.
>
> [...]
>
> A TCP **MAY** implement PUSH flags on SEND calls. If PUSH flags are not implemented, then the sending TCP: (1) must not buffer data indefinitely, and (2) **MUST** set the PSH bit in the last buffered segment (i.e., when there is no more queued data to be sent).
>
> [...]
>
> An application program is logically required to set the PUSH flag in a SEND call whenever it needs to force delivery of the data to avoid a communication deadlock. However, a TCP **SHOULD** send a maximum-sized segment whenever possible, to improve performance (see Section 4.2.3.4).

Da nicht am Ende jedes 4-KByte-Blocks das PSH-Flag gesetzt ist, muss man davon ausgehen, dass es nicht durch den Apache-Server, sondern direkt vom TCP-Stack (auf der Server-Seite) gesetzt wird. Damit sind nur die oberen beiden Ausschnitte relevant, so dass vermutlich vom TCP-Stack das PSH-Flag gesetzt wird, wenn eine voreingestellte Wartezeit ohne weitere Daten abgelaufen ist.

Bei den Frame-Nummern fällt auf, dass diese anfangs Abstand 2 haben (weil der Client jedes Paket bestätigt), später haben sie meist abwechselnd Abstand 1 und 2, weil der Client nur noch jedes zweite Paket bestätigt. Das liegt daran, dass der TCP-Stack das *Slow-start*-Verfahren [Ste97] verwendet.

Das verzögerte Paket Nummer 14 wird in den drei Frames 92, 93 und 95 übertragen:

```
Frame 92 (1454 bytes on wire, 1454 bytes captured)
Internet Protocol, Src Addr: 217.160.179.171 (217.160.179.171), Dst Addr: 192.168.1.1 (192.168.1.1)
Transmission Control Protocol, Src Port: http-alt (8080), Dst Port: 54187 (54187), Seq: 960368623,\
  Ack: 3097799097, Len: 1388
    Flags: 0x0010 (ACK)
Hypertext Transfer Protocol
    Data (1388 bytes)

Frame 93 (1454 bytes on wire, 1454 bytes captured)
Internet Protocol, Src Addr: 217.160.179.171 (217.160.179.171), Dst Addr: 192.168.1.1 (192.168.1.1)
Transmission Control Protocol, Src Port: http-alt (8080), Dst Port: 54187 (54187), Seq: 960370011,\
  Ack: 3097799097, Len: 1388
    Flags: 0x0010 (ACK)
```

```
Hypertext Transfer Protocol
    Data (1388 bytes)

Frame 95 (1386 bytes on wire, 1386 bytes captured)
Internet Protocol, Src Addr: 217.160.179.171 (217.160.179.171), Dst Addr: 192.168.1.1 (192.168.1.1)
Transmission Control Protocol, Src Port: http-alt (8080), Dst Port: 54187 (54187), Seq: 960371399,\
  Ack: 3097799097, Len: 1320
    Flags: 0x0018 (PSH, ACK)
Hypertext Transfer Protocol
    Data (1320 bytes)
```

Hier erfolgt also wieder eine Abtrennung, obwohl das folgende Paket Nummer 15 unverzögert ist. Der Zusammenhang zwischen Verzögerung und dem Versand von TCP-Paketen mit nicht maximaler Größe besteht also nur in einer Richtung: Bei größerer Wartezeit wird ein TCP-Paket verschickt, das die MSS nicht ausnutzt; andersherum kann man nicht folgern.

Störende Einflüsse des *Slow start* sind bei den Paketen 13 und 14 auszuschließen, da hier bereits die maximale Fenstergröße von 63848 Byte genutzt wurde.

Literaturverzeichnis

Eine Online-Version des Literaturverzeichnis' steht unter http://privat.hgesser.com/docs/Info-Diplom/ zur Verfügung.

Datumsangaben in Klammern kennzeichnen bei nur online verfügbaren Arbeiten das Datum, an dem der Zugriff auf die Web-Seite erfolgte.

[Amo94] Amoroso, Edward G.: *Fundamentals of Computer Security Technology*. Prentice-Hall, Inc., 1994. Erhältlich bei http://amazon.co.uk als Print-on-demand-Buch, £ 39,99.

[Ana] *Analyzer-Homepage*. http://netgroup-serv.polito.it/analyzer/ (29.12.2004).

[Apaa] *Download-Adresse für die Apache-Version 1.3.31*. http://archive.apache.org/dist/httpd/apache_1.3.31.tar.gz (29.12.2004).

[Apab] *Apache Web-Server*. http://www.apache.org.

[Apac] *Apache: HTTP Server Application Program Interface für Version 1.3*. http://httpd.apache.org/docs/misc/API.html.

[Bau03] Bauer, Matthias: *New Covert Channels in HTTP: Adding Unwitting Web Browsers to Anonymity Sets*. In: *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2003)*, Washington, DC, USA, Oktober 2003. http://www1.informatik.uni-erlangen.de/~bauer/109-bauer.ps.

[Bea99] Beazley, David M.: *Python Essential Reference*. New Riders Publishing, 1999. ISBN 0-7357-0901-7.

[BL73] Bell, D.E. und L. J. La Padula: *Secure Computer Systems: Mathematical Foundations*. Technischer Bericht, The MITRE Corporation, Bedford, MA, März 1973. Technical report, ESD-TR-73- 278-I.

[Bra89] Braden, R. T.: *Requirements for Internet hosts – communication layers*. RFC 1122, Internet Engineering Task Force, Oktober 1989. http://www.rfc-editor.org/rfc/rfc1122.txt.

[Cct] *Cctt-Homepage*. http://www.entreelibre.com/cctt/index_en.html (29.12.2004).

[Cha81] Chaum, David: *Untraceable electronic mail, return addresses, and digital pseudonyms*. Communications of the ACM, 4(2), Februar 1981. http://world.std.com/~franl/crypto/chaum-acm-1981.html.

[Com] *CommView-Homepage*. http://www.tamos.com/ (29.12.2004).

[CSo04] *Socket Programming in C: Programmbeispiele*, 2004. http://pont.net/socket/ (01.11.2004).

[Dae96] Daemon9: *Project Loki: ICMP tunneling*. Phrack, 7(49), 1996. http://www.phrack.org/show.php?p=49&a=6.

[Dae97] Daemon9: *LOKI2 (the implementation)*. Phrack, 7(51), 1997. http://www.phrack.org/show.php?p=51&a=6.

[DC03] Dyatlov, Alex und Simon Castro: *Exploitation of data streams authorized by a network access control system for arbitrary data transfers: tunneling and covert channels over the HTTP protocol*. Juni 2003. http://gray-world.net/projects/papers/html/covert_paper.html (12.06.2004).

[Eat97] Eaton, John W.: *GNU Octave – A high-level interactive language for numerical computations*. 3. Auflage, 1997. Benutzerhandbuch zu Octave.

[EL00] Ernst, Nico und Jörg Luther: *AOL/Netscape spioniert Surfer aus*. TecChannel, 2000. http://www.tecchannel.de/internet/468/ (12.06.2004).

[eth] *Ethereal-Homepage*. http://www.ethereal.com/.

[Eße05] Eßer, Hans-Georg: *Ausnutzung verdeckter Kanäle am Beispiel eines Web-Servers – Download-Bereich*, 2005. http://privat.hgesser.com/docs/Info-Diplom/.

[FBH+02] Feamster, Nick, Magdalena Balazinska, Greg Harfst, Hari Balakrishnan und David Karger: *Infranet: Circumventing Web Censorship and Surveillance*. In: *11th USENIX Security Symposium*, San Francisco, CA, August 2002. http://www.usenix.org/publications/library/proceedings/sec02/feamster/feamster.pdf.

[FGM+99] Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach und T. Berners-Lee: *Hypertext Transfer Protocol – HTTP*. RFC 2616, Internet Engineering Task Force, 1999. http://www.rfc-editor.org/rfc/rfc2616.txt.

[Gas88] Gasser, Morrie: *Building a Secure Computer System*. Van Nostrand Reinhold, Juni 1988. ISBN: 0442230222. http://nucia.ist.unomaha.edu/library/gasser.php.

[Ger02] Gerber, Tim: *Phone-home-Vorwürfe an Netscape*. Heise News, 2002. http://www.heise.de/newsticker/meldung/25493 (12.06.2004).

[Gla] *GLADE-Homepage*. http://glade.gnome.org/.

[GMS95] Goossens, M., F. Mittelbach und A. Samarin: *Der LaTeX-Begleiter*. Addison-Wesley, 1995.

[Gol03] Goltz, James P.: *Under the radar: A look at three covert communications channels*. GSEC practical assignment, 2003. http://web.mmert.org/~goltz/GSEC/Jim_Goltz_GSEC_edit.pdf (17.07.2004).

[Goo] *The Google Timeline*. http://www.google.com/corporate/timeline.html (18.02.2005).

[Ham50] Hamming, R. W.: *Error detecting and error correcting codes*. The Bell System Tech. Journal, XXIX(2):147–160, 1950. http://engelschall.com/~sb/hamming/ (eingescannt).

[Har01] Hartmann, Mike: *Netzwerk Sniffer*. TecChannel, 2001. http://www.tecchannel.de/hardware/766/ (12.06.2004).

[Het02] Hetzl, Stefan: *Steghide Man Page*, 2002. http://steghide.sourceforge.net/documentation/manpage.php (12.06.2004).

[Htt] *Httptunnel-Homepage*. http://www.nocrew.org/software/httptunnel.html (29.12.2004).

[Ile04] Ilett, Dan: *Lexmark accused of installing spyware*, November 2004. ZDNet UK, http://news.zdnet.co.uk/internet/security/0,39020375,39173517,00.htm (30.12.2004).

[JDB92] Jacobson, V., R. Draden und D. Borman: *TCP Extensions for High Performance*. RFC 1323, Internet Engineering Task Force, 1992. http://www.faqs.org/rfcs/rfc1323.html.

[Kes04] Kessler, Gary C.: *An Overview of Steganography for the Computer Forensics Examiner*. Forensic Science Communications, 6(3), 2004. http://www.fbi.gov/hq/lab/fsc/backissu/july2004/research/2004_03_research01.htm.

[Lam73] Lampson, Butler W.: *A note on the confinement problem*. Commun. ACM, 16(10):613–615, 1973. http://www.cs.cornell.edu/andru/cs711/2003fa/reading/lampson73note.pdf.

[Lan70] Lang, Serge: *Linear Algebra*. Addison-Wesley Publishing Company, 2. Auflage, 1970.

[LL99] Laurie, Ben und Peter Laurie: *Apache: The Definitive Guide*. O'Reilly, 2. Auflage, Februar 1999. ISBN: 1-56592-528-9.

[Lut96] Lutz, Mark: *Programming Python*. O'Reilly & Associates, Inc., 1. Auflage, 1996. ISBN 1-56592-197-6.

[Mac03] MacKay, David J. C.: *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003. http://www.inference.phy.cam.ac.uk/mackay/itila/.

[Mat96] Mathar, Rudolf: *Informationstheorie*, Band 9 der Reihe *Aachener Beiträge zur Mathematik*. Verlag der Augustinus Buchhandlung, 1996. Skript zur Vorlesung.

[MD90] Mogul, J. C. und S. E. Deering: *Path MTU discovery*. RFC 1191, Internet Engineering Task Force, November 1990. `http://www.rfc-editor.org/rfc/rfc1191.txt`.

[MM94] Moskowitz, Ira S. und Allen R. Miller: *Simple Timing Channels*. In: *SP '94: Proceedings of the 1994 IEEE Symposium on Security and Privacy*, Seite 56. IEEE Computer Society, 1994. `http://chacs.nrl.navy.mil/publications/CHACS/1994moskowitz-oakland.ps`.

[MP90] Mathar, Rudolf und Dietmar Pfeifer: *Stochastik für Informatiker*. Leitfäden und Monographien der Informatik. B. G. Teubner, Stuttgart, 1990.

[MSO01] Mitchell, Mark, Alex Samuel und Jeffrey Oldham: *Advanced Linux Programming*. Sams (Pearson Education), 2001. ISBN: 0735710430.

[Nma] *Nmap-Homepage*. `http://www.insecure.org/nmap/` (29.12.2004).

[Oct] *Octave-Homepage*. `http://www.octave.org/`.

[PAK99] Petitcolas, Fabien A. P., Ross J. Anderson und Markus G. Kuhn: *Information hiding—A survey*. Proceedings of the IEEE, 87(7):1062–1078, 1999. `http://www.petitcolas.net/fabien/publications/ieee99-infohiding.pdf`.

[PkZ] *PkZIP-Homepage*. `http://www.pkware.com/products/enterprise/unix/` (29.12.2004).

[Pos81a] Postel, John: *Internet Control Message Protocol*. RFC 792, Internet Engineering Task Force, September 1981. `http://www.rfc-editor.org/rfc/rfc792.txt`.

[Pos81b] Postel, Jon B.: *Internet Protocol*. RFC 791, Internet Engineering Task Force, September 1981. `http://www.rfc-editor.org/rfc/rfc791.txt`.

[Pos81c] Postel, Jon B.: *Transmission Control Protocol*. RFC 793, Internet Engineering Task Force, September 1981. `http://www.ietf.org/rfc/rfc793.txt`.

[Pos83] Postel, Jon B.: *TCP maximum segment size and related topics*. RFC 879, Internet Engineering Task Force, November 1983. `http://www.rfc-editor.org/rfc/rfc879.txt`.

[rev] *Revsh-Homepage*. `http://freshmeat.net/projects/revsh/` (29.12.2004).

[Sch95] Schneier, Bruce: *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., 1995.

[Sch97] Schweitzer, Frank: *Selbstorganisation und Information*. In: Krapp, H. und Th. Wagenbaur (Herausgeber): *Komplexität und Selbstorganisation – „Chaos" in Natur- und Kulturwissenschaften*, Seiten 99–129. Wilhelm Fink Verlag, 1997. http://summa.physik.hu-berlin.de/~frank/download/web-tueb.pdf.

[Sha48] Shannon, C. E.: *A Mathematical Theory of Communication*. The Bell System Technical Journal, 27:379–423, 623–656, Oktober 1948. http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf.

[Sko00] Skoll, David F.: *A PPPoE Implementation for Linux*. In: *Proceedings of the 4th Annual Linux Showcase & Conference*, Atlanta, Georgia, USA, Oktober 2000. USENIX. http://www.roaringpenguin.com/images/resources_files/PPPoEforLinux.pdf.

[SMC02] Shannon, Colleen, David Moore und K.C. Claffy: *Beyond folklore: observations on fragmented traffic*. IEEE/ACM Transactions on Networking, 10(6):709–720, Dezember 2002.
http://www.caida.org/outreach/papers/2002/Frag/frag.pdf.

[Smi02] Smith, Richard M.: *Serious privacy problems in Windows Media Player for Windows XP*. 2002. http://www.computerbytesman.com/privacy/wmp8dvd.htm (12.06.2004).

[Spa03] Spangler, Ryan: *Analysis of Remote Active Operating System Fingerprinting Tools*, Mai 2003. http://www.packetwatch.net/documents/papers/osdetection.pdf (12.06.2004).

[Sta02] Stallman, Richard M.: *Why Free Software is better than Open Source*. In: *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Oktober 2002.
http://www.fsf.org/philosophy/free-software-for-freedom.html.

[Ste97] Stevens, W. Richard: *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. RFC 2001, Internet Engineering Task Force, Januar 1997. http://www.faqs.org/rfcs/rfc2001.html.

[Ste00] Stevens, W. Richard: *Programmieren von UNIX-Netzwerken*. Carl Hanser Verlag, München, 2. Auflage, 2000. ISBN: 3-446-21334-1,
Beispielprogramme: ftp://ftp.cs.columbia.edu/pub/dcc/classes/CS4119-S98/stevens/unpv12e.tar.gz.

[Tou] Touretzky, Dave: *Steganography Wing of the Gallery of CSS Descramblers*. http://www-2.cs.cmu.edu/~dst/DeCSS/Gallery/Stego/ (29.12.2004).

[Xpr] *Xprobe2-Homepage*. http://www.sys-security.com/index.php?page=xprobe.

[Yes] *YesCoder-Homepage*.
http://mitglied.lycos.de/JoergGrohne/yescoder.html.

Stichwortverzeichnis

Fett hervorgehobene Seitenzahlen im Stichwortverzeichnis weisen auf eine Definition in Kapitel 2 hin; Einträge in `nichtproportionaler Schrift` beziehen sich auf Programme, Funktionen und Variablen.

T

U

V

W

X

Z

Zitate